SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA

Faculty of Mechanical Engineering

Reg. No.: SjF-104462-81339

FloatShield: An Educational Device for Air Levitation Experiments

Master's thesis

2020 **Bc. Peter Chmurčiak**

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA

Faculty of Mechanical Engineering

Reg. No.: SjF-104462-81339

FloatShield: An Educational Device for Air Levitation Experiments

Master's thesis

Study programme:	Applied Mechanics and Mechatronics
Study field:	Cybernetics (5.2.16. Mechatronics)
Training workplace:	Institute of Automation, Measurement and Applied Informatics
Thesis supervisor:	doc. Ing. Gergely Takács, PhD.
Consultant:	Ing. Erik Mikuláš

Bratislava, 2020 **Bc. Peter Chmurčiak**

Slovak University of Technology in Bratislava
Institute of automation, measurement and applied informatics

Faculty of Mechanical Engineering
Academic year: 2019/2020
Reg. No.: SjF-104462-81339

STU SjF

MASTER THESIS TOPIC

Student:	**Bc. Peter Chmurčiak**
Student's ID:	81339
Study programme:	Applied Mechanics and Mechatronics
Study branches combination:	Mechanical Engineering, Cybernetics
Thesis supervisor:	doc. Ing. Gergely Takács, PhD.
Consultant:	Ing. Erik Mikuláš
Workplace:	IAMAI, FME, STU in Bratislava

Topic: **FloatShield: An Educational Device for Air Levitation Experiments**

Language of thesis: English

Specification of Assignment:

The task of the student and the goal of the thesis is to design and implement changes in the software and hardware of an expansion module designed for the Arduino microcontroller prototyping boards. Specifically, it is an extension module called FloatShield, which allows one to perform experiments in levitating an object – such as a ball – in a stream of air.

As part of the thesis work, the student should
- introduce the FloatShield original concept with regard to similar existing equipment in the literature,
- design and implement hardware changes for the FloatShield module,
- write an application programming interface (API) for device control in C / C ++ for the Arduino IDE, for MATLAB and for Simulink,
- mathematically describe a model of the device dynamics based on physical principles,
- perform system identification experiments and identify unknown model parameters, verify the model against acquired data,
- design, create and describe didactic examples on the proportional integral derivative (PID), linear quadratic (LQ) and linear model predictive control (MPC) of the ball position,
- interpret the achieved results and compare the algorithms used in terms of use for embedded systems.

Length of thesis: 50-60 p.

Assignment procedure from: 17. 02. 2020

Date of thesis submission: 19. 06. 2020

Bc. Peter Chmurčiak
Student

prof. Ing. Cyril Belavý, CSc.
Head of department

prof. Ing. Peter Šolek, CSc.
Study programme supervisor

Declaration on word of honour

I hereby declare that I am the sole author of this master's thesis and that I have not used other sources than those listed in the bibliography and identified as references.

Bratislava, 18. June 2020 .....

Signature

I would like to express my sincere gratitude to my advisor doc. Ing. Gergely Takács, PhD. for his patience, enthusiasm and immense knowledge combined with a kind approach. His guidance helped me to overcome obstacles that occurred. I would also like to thank Ing. Erik Mikuláš and doc. Ing. Martin Gulan, PhD., who with their helpful advice and critical thinking, guided me when needed.

Bratislava, 18. June 2020 Bc. Peter Chmurčiak

Názov práce: FloatShield: Výučbový prístroj pre experiment levitácie v prúde vzduchu
Kľúčové slová: Arduino, štít, TOF, levitácia, riadenie, API
Abstrakt: Pre našu modernú spoločnosť je nevyhnutné neustále zlepšovať používané postupy v oblasti vzdelávania. Udržateľnosť budúceho rastu vo veľkej miere závisí od kvality odbornej prípravy v súčasnosti poskytovanej inžinierom a vedcom. Dopyt po dostupných akademických nástrojoch ktoré by mohli byť prostriedkom na pokrok vo vyučovacom procese neustále narastá. V tejto práci je rozoberaný jeden z takýchto možných nástrojov. Cieľmi práce sú: predstavenie prístroja FloatShield, návrh a výroba jeho nasledujúcej verzie, vybudovanie potrebnej softvérovej podpory vo vybraných programovacích prostrediach a vykonanie experimentov riadenia na prístroji použitím rôznych algoritmov. Práca začína načrtnutím konceptu tohto aparátu pričom poskytuje odkazy na podobné zariadenia. Následne sú opísané detaily jeho pôvodného hardvéru a DPS, kde sú tiež predložené ich možné zmeny a vylepšenia. Na ich základe je potom navrhnutá a vytvorená ďalšia verzia prístroja. Ďalej nasleduje opis jednotlivých metód (funkcií) vytvorených pre zariadenie v prostrediach Arduino IDE, MATLAB a Simulink. V ďalších častiach sú podrobne opísané procesy modelovania systému, identifikácie systému a odhadu stavov systému pomocou Kalmanovho filtra. Na koniec sú uskutočnené experimenty riadenia (použitím príslušných metód) využitím algoritmov PID, LQR a MPC, vo všetkých troch vyššie uvedených prostrediach.

Title: FloatShield: An Educational Device for Air Levitation Experiments
Keywords: Arduino, shield, TOF, levitation, control, API
Abstract: It is crucial for our modern society, to continuously enhance its practices in the realm of education. Sustainability of the future growth, heavily depends on the quality of training, provided to the engineers and scientists in the present. There is currently a growing need for available academic tools, which could be a means of advancement in the teaching process. One of such tools is presented in this work. The aims of the thesis are to: introduce the FloatShield device; design and create its next version; build the necessary software support in selected programming environments; perform control experiments on the device using various algorithms. The thesis starts by outlining the concept of the apparatus, while providing references to similar ones. Afterwards, the original hardware and PCB are described, and possible enhancements are proposed. Then, the next version of the device is designed and created, while keeping those propositions in mind. This is followed by introduction of individual methods (functions), built for the device within the Arduino IDE, MATLAB and Simulink environments. Subsequently, the processes of system modeling, system identification and Kalman filter estimation, are described in detail. Finally, control experiments are conducted (using the respective methods) utilizing PID, LQR and MPC algorithms, within all three aforementioned environments.

Contents

Acronyms

ADC	analog-to-digital converter.
API	application programming interface.
APRBS	amplitude modulated pseudo-random binary sequence.
BLA	Basic Linear Algebra.
CAD	computer-aided design.
DAQ	data aquisition.
EMF	electromotive force.
GUI	graphical user interface.
I/O	input/output.
I2C	inter-integrated circuit.
IDE	integrated development environment.
LQR	linear quadratic regulator.
MCU	microcontroller unit.
MIMO	multiple-input multiple-output.
MOSFET	metal-oxide-semiconductor field-effect transistor.
MPC	model predictive control.
OOP	object-oriented programming.
PCB	printed circuit board.
PID	proportional-integral-derivative (controller).
PLC	programmable logic controller.
PRBS	pseudo-random binary sequence.
PWM	pulse width modulation.
RPM	rotations per minute.

SBC	single-board computer.
SCL	synchronous clock.
SDA	synchronous data.
SISO	single-input single-output.
TOF	time of flight.
USB	universal serial bus.

List of Symbols

$\mathbf{A}$	System matrix of a discrete state-space model.
A_b	Surface area of the ball exposed to the air flow.
$\mathbf{A}_c$	Matrix multiplying the optimization variable on the left side of the constraint in the form of the inequation $\mathbf{A}_c \overrightarrow{\mathbf{u}}_{(k)} \leq \mathbf{b}_c$.
$\widetilde{\mathbf{A}}$	System matrix of a discrete state-space model extended by the integrator state.
$\mathbf{b}_0$	Matrix on the right side of the constraint in the form of the inequation $\mathbf{A}_c \overrightarrow{\mathbf{u}}_{(k)} \leq \mathbf{b}_0 + \mathbf{B}_0 \mathbf{x}_{(k)}$.
$\mathbf{b}_c$	Column vector on the right side of the constraint in the form of the inequation $\mathbf{A}_c \overrightarrow{\mathbf{u}}_{(k)} \leq \mathbf{b}_c$, or in the equation $\mathbf{b}_c = \mathbf{b}_0 + \mathbf{B}_0 \mathbf{x}_{(k)}$.
$\mathbf{B}$	Input matrix of a discrete state-space model.
$\mathbf{B}_0$	Matrix on the right side of the constraint in the form of the inequation $\mathbf{A}_c \overrightarrow{\mathbf{u}}_{(k)} \leq \mathbf{b}_0 + \mathbf{B}_0 \mathbf{x}_{(k)}$, which is multiplied by the current state.
$\widetilde{\mathbf{B}}$	Input matrix of a discrete state-space model extended by the integrator state.
$\mathbf{C}$	Output matrix of a discrete state-space model.
C_D	Drag coefficient.
$\mathbf{D}$	Direct transition matrix of a discrete state-space model.
$\delta u_{(t)}$	Perturbation of system input from equilibrium at time t.
$\delta \mathbf{x}_{(t)}$	Perturbation of system state vector from equilibrium at time t.
$\delta H_{(s)}$	Perturbation of ball position from equilibrium expressed in the s-domain.
$\delta U_{(s)}$	Perturbation of system input from equilibrium expressed in the s-domain.
$\delta V_{(s)}$	Perturbation of air velocity from equilibrium expressed in the s-domain.
$e_{(k)}$	Error at discrete time k.
$\mathbf{e}_{(k)}$	Error vector at discrete time k.

$\mathbf{F}$	MPC cost function matrix independent of the input sequence $\overrightarrow{\mathbf{u}}_{(k)}$.
$\mathbf{F_c}$	System matrix of a continuous state-space model.
$F_{D(t)}$	Drag force at time t.
F_G	Gravitational force.
g	Gravitational acceleration.
$\mathbf{g_r}$	Gradient, where $\mathbf{g_r}^{\mathrm{T}} = \mathbf{x}_{(k)}{}^{\mathrm{T}}\mathbf{G}^{\mathrm{T}}$.
$\mathbf{G}$	MPC cost function matrix, where $\mathbf{g_r}^{\mathrm{T}} = \mathbf{x}_{(k)}{}^{\mathrm{T}}\mathbf{G}^{\mathrm{T}}$.
$\mathbf{G_c}$	Input matrix of a continuous state-space model.
$h_{(t)}$	Position of the ball inside the tube at time t.
$\mathbf{H}$	Hessian matrix.
$\mathbf{I}$	Identity matrix.
$J_{(k)}$	Value of the cost function at discrete time k.
k	Discrete-time index, where $t = kT_s$.
$\mathbf{K}$	Gain matrix of the linear quadratic regulator.
K_f	Fan gain constant.
K_p	Proportional gain in PID.
$\widetilde{\mathbf{K}}$	Gain matrix of the linear quadratic regulator extended by the integrator state.
m	Mass of the ball.
n_p	Prediction horizon.
$\mathbf{P}$	Terminal state penalty matrix.
$\mathbf{P}_K$	Estimation error covariance matrix.
$\mathbf{Q}$	State penalty matrix.
$\mathbf{Q}_K$	Process noise covariance matrix.
$\widetilde{\mathbf{Q}}$	State penalty matrix extended by the integrator state.
$\mathbf{r}_{(k)}$	Reference vector at discrete time k.
$\mathbf{R}$	Input penalty matrix.
$\mathbf{R}_K$	Measurement noise covariance matrix.
$\widetilde{\mathbf{R}}$	Input penalty matrix extended by the integrator state.
ρ	Density of the air.
s	Laplace operator.
t	Continuous time.

T_d	Derivative time in PID.
T_i	Integral time in PID.
T_s	Sampling period.
τ_1	System time constant.
τ_2	Fan time constant.
$u_{(k)}$	System input at discrete time k.
$\mathbf{u}_{(k)}$	System input vector at discrete time k.
$\overrightarrow{\mathbf{u}}_{(k)}$	Sequence of future inputs calculated at discrete time k.
$\overrightarrow{\mathbf{u}}^*_{(k)}$	Optimal sequence of future inputs calculated at discrete time k.
$v_{(t)}$	Velocity of the air inside the tube at time t.
$\mathbf{x}_{(k)}$	System state vector at discrete time k.
$\mathbf{x}_{r(k)}$	Reference state vector at discrete time k.
$\hat{\mathbf{x}}^-_{(k)}$	A priori estimate of the system state vector at discrete time k.
$\mathbf{x}^{\mathbf{I}}{}_{(k)}$	Integrator state at discrete time k.
$\hat{\mathbf{x}}_{(k)}$	A posteriori estimate of the system state vector at discrete time k.
$\widetilde{\mathbf{x}}_{(k)}$	State vector extended by the integrator state at discrete time k.
$y_{(k)}$	System output at discrete time k.
$\mathbf{y}_{(k)}$	System output vector at discrete time k.

Introduction

The buzzword of recent decade is "sustainability" of technological and economical growth. Although the idea of everlasting growth is extremely appealing, advancement causes immense pressure on resources, which become scarcer and scarcer. The only way how to resolve the problem of balance between continuous growth and availability of resources, is the focus on efficiency in all areas.

Therefore, development of advanced technologies must go hand in hand with development of modern teaching methods and practices, to be used in education of future engineers, scientists and experts. After all, it will be their knowledge and skills that the further enhancement and management of those technologies will ultimately depend on.

In an ideal world, theoretical insight gained by students in lecture halls would be accompanied by practical experiments in a laboratory environment, through which everyone could verify the validity and indisputable usefulness of the learned theory. The knowledge thus proven by experience, has bigger potential to be much more easily stored in the memories of individuals, than if one part of an inseparable pair of theory-practice, is absent in the learning process.

However, achieving this kind of experiential teaching is rarely that simple in reality, as modern technologies are often heavily dependent on up-to-date equipment, that in most cases consists of sensitive, fragile, and above all, very costly devices. Of course, there are commercial companies on the market, that offer professional laboratory aids to universities and other educational institutions, to teach about various mechatronic systems. Such tools are made of high quality components, and in addition to reliable software support, they are often provided with extensive documentation for teachers. It could almost seem like an ideal solution to the growing need for such teaching aids, until the already mentioned high price of those devices is taken into consideration. The cost often in the range of tens of thousands of euros for one unit is an issue, that makes equipping even a single laboratory with several such devices an unrealistic and unfeasible feat for most institutions. Even in the case cost was not an issue, such delicate apparatus could be used by students only under strict supervision, what would highly restrict possibilities of its educational usage.

Despite the very limited supply, the demand for similar teaching tools is steadily increasing, and because of that, many classrooms resort to designing and creating their own devices. Unfortunately, in almost all cases, those often fully functioning prototypes remain accessible only to a limited group of people—its creators. Because of that, there is no room for its wider usage, development and potential cooperation between students and teachers belonging to different institutions.

But it does not have to end like this—the young, non-profit, open-source project with a name AutomationShield is a living proof of how the functioning prototype can be elevated into something more, through sharing.

Chapter 1

FloatShield Origins

1.1 AutomationShield

The AutomationShield project is dedicated to the creation of affordable, but well-made educational tools, that are currently desperately needed. To achieve this goal, the project has chosen to use the Arduino—an open-source microcomputer prototyping platform. Thanks to its user-friendliness and wide scope of application, the Arduino is already being used by many universities around the globe to teach microcomputer technology. The selection of Arduino platform for this purpose also stemmed from the fact, that all most commonly used Arduino boards have adopted the same standardized pin layout, also known as the Arduino R3. This greatly increased the accessibility of the functionality-expanding modules—widely known under the name "shields".

As its name partly suggests, the project deals with the development of the so-called shields, whose main function is to—by usage of various components and accessories—extend the base functionality of the Arduino microcontroller unit (MCU), and thus enable it to serve more particular and specialized purposes. At its core, a shield is nothing more than a printed circuit board (PCB) containing electrical pathways, that interconnect components soldered onto it. In general, there are numerous types of shields, with large variety of purposes—some allow Arduino boards to communicate wirelessly with other devices, other allow them to take in information from a number of different sensors, while others can facilitate control of various electric actuators.

However, shields developed by the AutomationShield are different from the aforementioned, commonly available ones. Each of its shields is unique—specialized to enable a certain type of physico-mechanical experiments—often aimed at teaching about methods of control of mechatronic systems. In addition to that, it also has a great potential to serve as an object of scientific work. This fact is evidenced by recently published scientific papers about: the so-called HeatShield, enabling experiments with thermal control of a 3D printer hot end [1]; the OptoShield, enabling a low cost optical feedback experiment [2]; the MagnetoShield, enabling experiments concerning magnetic levitation [3]; the LinkShield, enabling rotary flexible link experiments [4].

The great idea behind these shields, in addition to their Arduino R3 pinout compatibility and therefore full compatibility with the most widely used models of Arduino boards is, that all the components required for carrying out the various experiments and measurements are already contained within the PCB—the shield body.

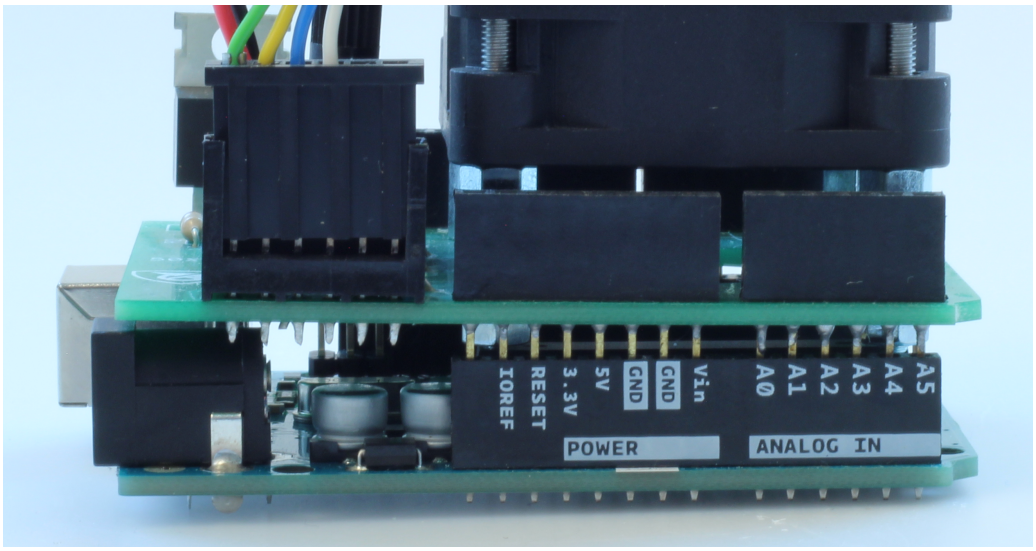


Figure 1.1: Illustration of limited size of the shield (side view).

Due to the standardized dimensions of the Arduino R3 pinout, the shield bodies have to be of limited size to allow proper mounting on the board. That can also be seen as an advantage, as the common benefit of compact dimensions is relatively low production price, moving in range of at most a few tens of euros. For example, a single unit of one of the first shields in the AutomationShield project, the already mentioned OptoShield, can be fully manufactured for a price of less than 3€ [2].

In addition to the shields themselves, the project also manages the design of related software tools. These are most often realized through the libraries containing a number of predefined functions. By including those libraries in their preferred integrated development environment (IDE), the users can choose from the list of functions according to their specific needs and goals.

As it was already mentioned, this project is categorized as open-source, what in reality means that all PCB drawings, documentation and software support are publicly available through the project website to all interested individuals and institutions. The public availability of this crucial information opens up new possibilities for mutual cooperation, involving knowledge and the transfer of results, and thus allowing further development and enhancement of the devices themselves and their related software.

The openness and public accessibility of the project is a proven way (even by the Arduino platform itself) of how both the project and its users can benefit from mutual cooperation.

Currently there are ten shields being developed and maintained within the AutomationShield project. Each one of them is unique, created using different electromechanical components and utilizing different physical principles.

The primary type of experiment possible to conduct with each shield can be often guessed by its very name, specifically by the first part. One of those shields is the so-called "FloatShield", which will be further explored in detail in this thesis.

1.2 Shield Versioning

Before going any further, it is important to mention, that each shield is in and of itself a prototype with its quirks and flaws. By its usage, new ideas for betterment arise, what is of course expected, but it would be impractical to realise each new idea right away, as most of them require change in the hardware of the device. Change in hardware, is therefore realised only when a sufficient number of good ideas has been accumulated—enough to justify the process of developing and manufacturing a new PCB, through which the shield is mounted onto the Arduino board.

The progress made on hardware parts of each shield can be then divided into separate releases with the idea, that each newer release brings, or at least should bring, improvements to the previous version. Each shield then, may have multiple releases differentiated by a marking on their physical bodies, consisting of the letter "R" (as in the word "Release"), followed by a natural number representing the order in which this release was created (for example the first release R1 will be followed by second release R2).

Same method of differentiation is applied within the software environment of each shield, often implemented through the definition of a preprocessor token—thus enabling a single library to serve all releases.

Each shield release can then be considered as a variation of the same device, often with significant changes and enhancements.

1.3 The Concept of FloatShield

The first part of the shield name describes its area of application. Based on the name "FloatShield", it could be correctly assumed that this specific shield will have something to do with floating. If there was a definition of FloatShield in a dictionary, it would probably state something similar to: FloatShield is a device that enables its user to conclude aerodynamic experiments. In our case, the definition would be extended by: experiments concerning a position control of a floating object inside a transparent tube through the regulated stream of air.

Object floating in a stream of air is a generally known experiment around the world, possibly thanks to its playful and visually appealing nature. An air levitation device, allowing such experiment, can most commonly be seen in laboratories designated to teach control engineering and mechatronics in higher education. Despite its simple design, it hides some unexpected pitfalls concerning, among others, nonlinear system behaviour, what is probably the reason why it has been an object of interest of many scientific papers.

There are some possible variations, but at its core, every device allowing this type of experiment consists of:

- **Floating object**—in most cases in the shape of a sphere [5–15], but it may depend on other factors, such as if the floating object is enclosed within some form of air tunnel [6–10, 12, 13, 15], or is floating freely by utilizing the Coandă effect [5, 11] to limit horizontal movement, as the shape drastically impacts the amount of drag produced [16] and influences object stability within the turbulent flow. Other possible shapes worth considering include cylinder, cone or a droplet.

As for the material selection, the object is often made of light materials such as cork, cotton, or polystyrene, depending on the power of used air source, and potential requirements on mechanical properties. There is also a possibility of using 3D printing technologies for more specific shapes, but in these cases it is important to keep the object weight to minimum.

- **Source of air flow**—depending on the size of the device, either an air compressor, air blower [5, 11, 14], or air fan [6–10, 13, 15], can be used to produce the required stream of air. The choice is made based on the intended purpose of the device, for example, whether the device will be movable or stationary—whether the issues of size and power supply need to be addressed or there are no such limitations.

  Other factors contributing to the selection process include the required performance levels of the source, based on the amount of resistance it will have to overcome (influenced by object weight, object shape, air tunnel cross sectional area) and, required sensitivity of power control—how quickly should the source be able to change its power level and therefore air output.

- **Means of monitoring object position**—the most common approach of locating the floating object is by using distance sensors. In the majority of cases, either infrared [5–7, 9, 14, 15], ultrasonic [8, 9, 13], or a laser distance sensor [10], is fixed near the trajectory of the expected movement of the object, in a way as to not restrict its movement while still being able to "see" it.

  The utilized type of sensor depends mostly on the required speed and accuracy with which the object movement has to be followed, but also on specific conditions—for example laser sensors are ideal for detecting even very subtle and quick movements, but may experience problems with reflective surfaces in their proximity and are oftentimes expensive. On the other hand, ultrasonic sensors are not able to measure the distance with such minuscule precision and often have restrictions of minimal and maximal possible measured distance, but are much cheaper and do not experience problems with reflectivity.

  It is also possible, to use video cameras for this purpose [11, 12], although it might be considered the most expensive and complicated way of its realization, as there is an additional need for costly video capturing hardware and image processing software—if hard real-time constraints are to be enforced.

- **Control unit**—based on the intended application of the device, there are several methods of information management and control—for stationary devices the better way might be using a personal or industrial computer equipped with either a data aquisition (DAQ) device [5, 10, 12], programmable logic controller (PLC) [11], or some other type of external controller, through which the computer gathers data and calculates required system input. For movable and more compact devices the better way might be using embedded device such as microcontroller unit (MCU) [8] or single-board computer (SBC) [7, 17] as the "brain". Yet another way might be a combination of both—using MCU as the bus through which all the sensors are connected to the computer, where the calculations take place [13, 14]. This is also

more suited for stationary devices, but it is less expensive, as the sensors compatible with MCU are much cheaper than their DAQ compatible counterparts.

With each of these methods, the realisation of the device has to be different, since the personal computer requires a specific software—for example Simulink or LabView—and specific hardware—such as DAQ device and compatible sensors. Same goes for the MCU and SBC—they each have their own software and their own sensors that can be used with them.

Each method has its advantages and disadvantages—for example the MCU is very compact and relatively cheap, but it is limited in memory and in processing speed, while DAQ device can process large amounts of data from various sensors very quickly, but is quite expensive.

Of course, there can be differences in the design of devices and in materials used, but these four things are the main defining parts, and need to be chosen with the purpose in mind. These choices will influence size, price and therefore the availability of the final device.

Until now, such devices were rarely meant to be freely available, so the issue of availability was not important. With AutomationShield, this problem becomes actual—to serve the widest possible audience while maintaining a high standard, the balance between the affordability and complexity of the devices has to be constantly maintained.

One of such devices was, and still is the FloatShield.

1.4 FloatShield R1

The first FloatShield hardware was designed at the Faculty of Mechanical Engineering of the Slovak Technical University in Bratislava, within a student semestral project for the subject named Microprocessors and Microcomputers. The project was concluded with the creation of the first release—named FloatShield R1.

Its original creators had only one semester to design, manufacture and present a working prototype, what took a lot of time and effort. They were successful in building the hardware, however, there was not enough time to build any extensive software support for the device—that became the task of their successor and one of the main goals of this thesis.

The hardware part of the FloatShield R1 can be considered to be the starting point of this work.

1.4.1 R1 Hardware

The body of the first release of FloatShield—FloatShield R1—along with its rendered CAD model image is depicted in Fig. 1.2 while Fig. 1.3 displays a detailed view of the lower and upper part of the device.

An Arduino board (a) acts as a base of the whole apparatus. It is important to note, that not every Arduino board can be used with the FloatShield R1—there are two requirements that need to be simultaneously met: the board has to be R3 pinout compatible and, at the same time its MCU has to utilize 5 V operating voltage.

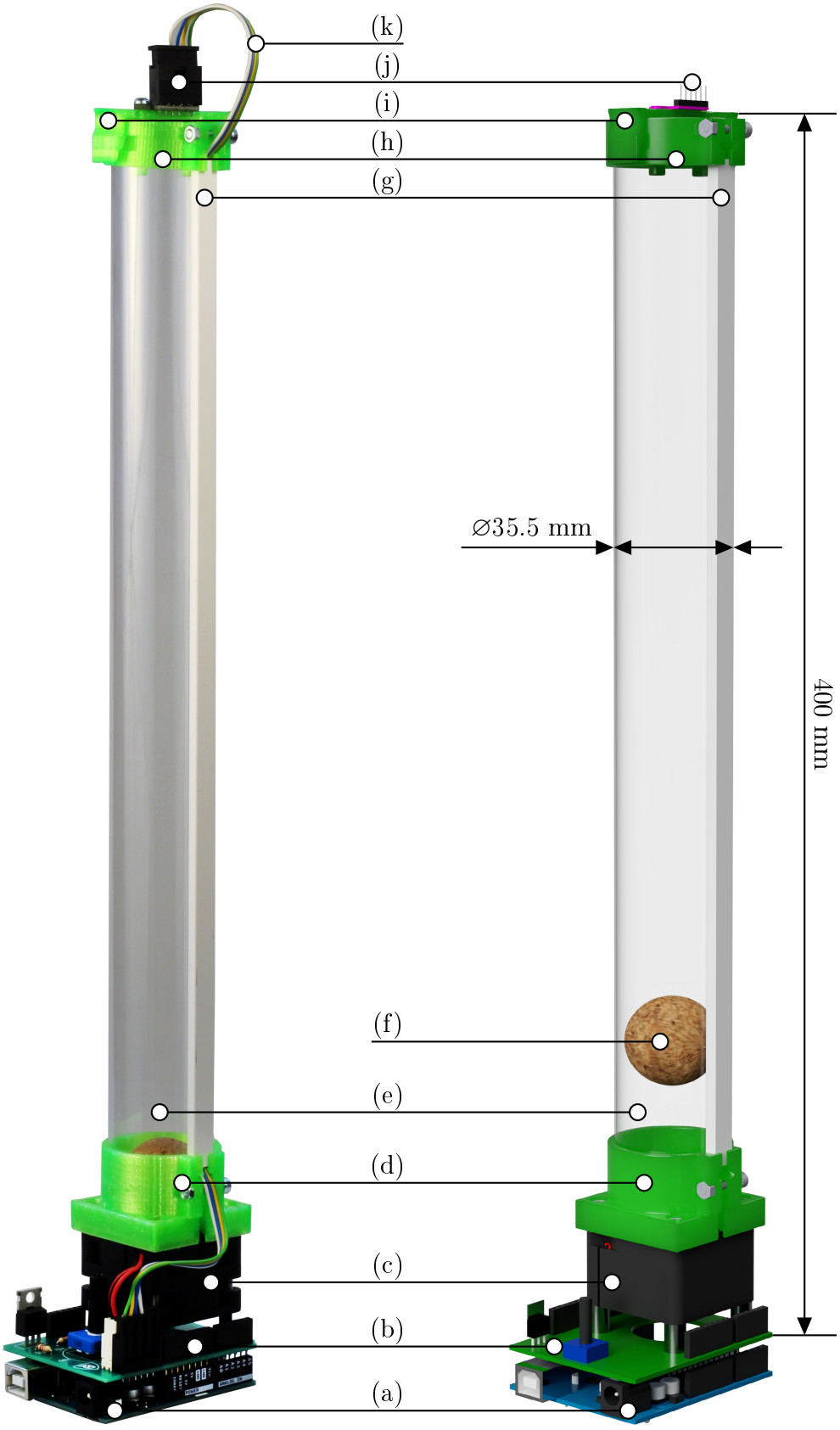


Figure 1.2: Photograph of the FloatShield R1 (left) and its CAD render (right).

When those two conditions are satisfied by a specific Arduino board, it can be used with the FloatShield R1. The two most common boards that fit into this category and therefore can be used with the FloatShield R1 are Arduino UNO and Arduino MEGA2560.

By using stackable header pins (p), that provide both mechanical and at the same time electrical connection between stacked segments, the shield (b) is mounted onto the Arduino board (a). The connection is possible, because the standardized R3 pinout is used both by the board and the shield—the stackable headers are located in the same positions and therefore fit into each other. As it has been previously mentioned, the shield (b) is basically a PCB that interconnects all the electrical components that are soldered onto it, through conductive pathways—including the stackable header pins (p).

The DC electric axial fan (c) is bolted directly into the shield (b) by the help of four hexagonal spacers. This ensures, that the fan (c) stays in its place despite the constant vibrations that are a natural by-product of its activity.

A 3D printed tube clamp (d) is placed onto the fan, whose main purpose is to provide a tight connection between the fan (c) and the tube (e), as they have different shapes of cross sections and do not fit naturally.

The biggest part of the apparatus is the approximately 340 mm long plastic translucent tube (e) with a 35.5 mm diameter. Its translucency provides better visual experience for the user—the behaviour of the ball (f) can be thus observed directly. The main purpose of the tube (e) is to constrain horizontal movements of the ball (f), while also limiting the loss of power of the air flow that would naturally occur, with increasing distance from its source (c).

A cork ball (f) with a diameter of 30 mm moves inside the tube (e). It has a shape of almost perfect sphere and thanks to the low density of cork, it weighs only 3.84 g. While the ball (f) moves primarily in the vertical direction, because of the roughly 5 mm difference between inner diameter of the tube (e) and outer diameter of the cork ball (f), there is a room for small horizontal movements often occurring in the form of oscillation.

A plastic U-profile (g) runs alongside the tube (e), with the goal of providing guidance and protection to the electric wires (k), that allow communication between the shield at the bottom (b) and the distance sensor (j) at the top of the device. At the lower end, it is held in its position with a 3D printed clamp (d) and at the upper end with a 3D printed flange (h).

The flange (h) is fitted onto the upper end of the tube (e) and serves as a place of attachment for the U-profile (g), and at the same time for a 3D printed sensor holder (i).

The distance sensor (j) is bolted onto the 3D printed sensor holder (i) in a way, that they form a single piece. At the middle of the holder (i), right under the sensor (j), there is a rectangular hole to allow the sensor (j) to "see" through it. There is a shape joint between the two 3D printed parts—the holder (i) has a small protrusion while the flange has a small grove (h) that fit together—so that the parts can be connected or disconnected by sliding. The advantage of mounting the sensor (j) with its holder (i) onto the flange (h) this way, is the ability to easily mount and dismount it, by simply sliding in the horizontal direction, and therefore gain access to the ball (f) in case it has to be replaced or modified. Otherwise, it would be necessary to disassemble the whole upper part to gain access to the inside of the tube (e).

All the 3D printed parts (d),(h),(i) were created with the Prusa i3 MK3 printer, and their CAD models and printing files can be found on the project website.

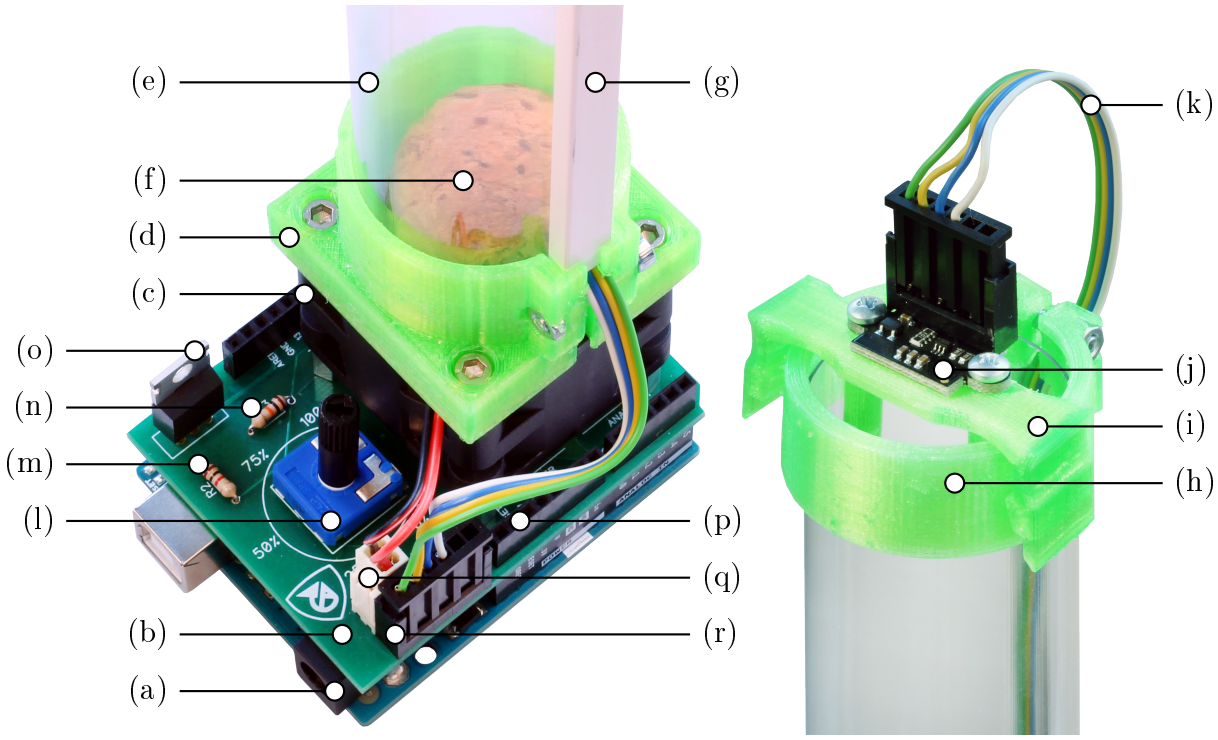


Figure 1.3: Detailed view of the lower and the upper part of the device.

One may find the arguably most important part at the top of the apparatus—the VL53L0X time of flight (TOF) laser ranging sensor from ST Microelectronics. Based on its description provided by the manufacturer, despite its very compact dimensions it is able to accurately measure absolute distances up to 2 metres, with no regard to measured object reflectance, unlike conventional technologies. It utilizes inter-integrated circuit (I2C) interface for communication and data transfer, and therefore the sensor (j) has to be connected to the I2C interface bus of the Arduino board (a)—specifically to the synchronous data (SDA) and synchronous clock (SCL) ports that are intended for I2C communication. Thanks to the fact that the sensor is integrated into a convenient open-source breakout board (j), the signal can be transferred from corresponding pins of (j) with cables (k) through the connector (r) to the shield (b), which then brings the signal through the electric pathways to the I2C bus of the Arduino board (a).

The scheme of the FloatShield R1 electric circuitry can be found in Fig. 1.4, with most of the connected components also visible in Fig. 1.3.

Within the schematic, the electrical connections between individual components and Arduino board (a) are displayed, where TOF sensor (j) and fan (c) are represented only by their connectors (r)(q).

The TOF (j) sensor draws current from `5V` pin through its connector (r), and is grounded by the `GND` pin. To be able to communicate with Arduino board (a), it is additionally linked to the Arduino I2C bus pins (`SDA` and `SCL`). Its connections with the `D7` and `D8` pins turned out to be redundant for the ordinary user, and are not implemented in this application. However, these can be utilized to set up some advanced functionalities of the TOF distance sensor (j).

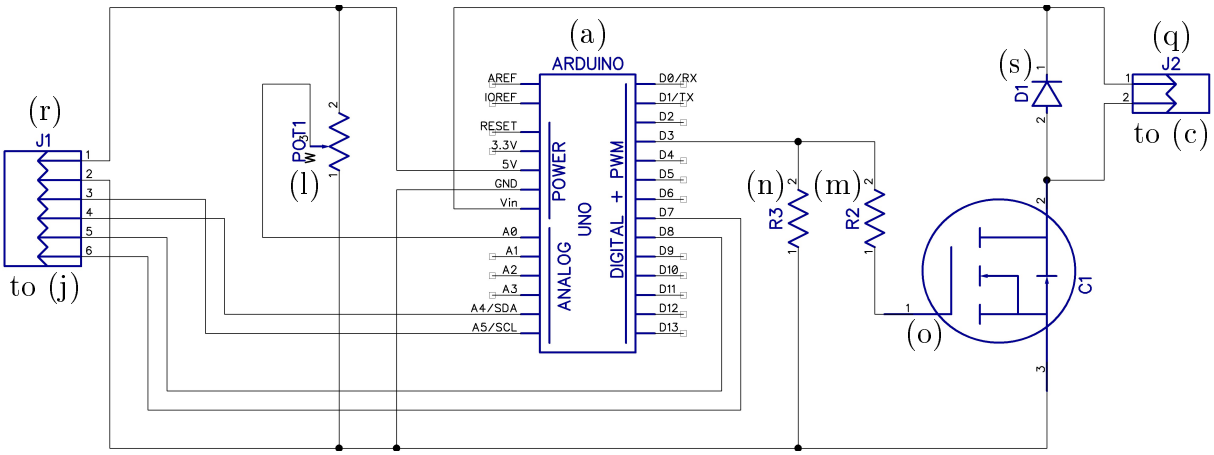


Figure 1.4: Electrical circuit schematic of the FloatShield R1.

One may find a potentiometer runner (l) in the centre of the user accessible part of the shield. It is powered from the `5V` pin and grounded by the `GND` pin. The input voltage reduced by the position-dependent resistance of the potentiometer (l), is then being read through the `A0` pin that utilises the analog-to-digital converter (ADC) capabilities of the Arduino board (a).

The fan (c) has only two connectors—for power and for ground—while its required operating voltage is 12 V. As the Arduino board (a) powered through USB cable is capable of providing only 5 V output and only a limited amount of current, a separate power supply is needed to run and operate the device—a 12 V wall adapter with a jack connectable to the Arduino board. With the adapter connected, the required voltage is supplied through the `VIN` pin on the board to the fan (c), by connecting it through the fan connector (q) to the shield (b). However, due to the fact that there are only two connectors to the fan (c), its power output can not be reasonably controlled without the use of an analog signal (that the Arduino board itself is not capable of creating).

Because of that, a different method of controlling power output had to be devised, and a suitable component able to fulfill that purpose turned out to be a transistor (o). Specifically, an N-channel metal-oxide-semiconductor field-effect transistor (MOSFET) (o) that acts as a very fast switch for the incoming current going to the fan (c). This component opens and closes the circuit between the `VIN` pin, fan (c) and grounding pin `GND`. The deciding factor on whether the circuit is closed or open is the voltage level on the `GATE` pin of the MOSFET that is marked with number 1 on its schematic symbol (o). To put it simply, if there is a sufficient voltage on the `GATE` pin, the current can flow freely through the MOSFET (thus the circuit is closed), while if there is no or insufficient voltage on the `GATE` pin, no current can flow through it (circuit is open).

The `GATE` pin is connected through a 1 kΩ resistor (m), whose purpose is limiting current draw, to the `D3` pin that is capable of pulse width modulation (PWM). To ensure that the voltage provided to the `GATE` pin is provided only intentionally, the 10 kΩ pull-down resistor (n) is added, that provides grounding to the `GATE` when the `D3` pin output should be "zero".

The signal created with PWM closely resembles a clock signal or square wave, with the difference, that the width of the squares, also called the duty cycle, can be continually

modified. Such a signal connected to the `GATE` pin of the MOSFET then provides a reliable way of controlling the power output of the fan, by modifying the amount of time the fan is "turned on" and "turned off" within the signal period.

Finally, a diode (s) is added to the fan (c) circuit, to provide protection against back electromotive force (EMF) caused by the electric motor within the fan (c).

The top and bottom side of the FloatShield R1 PCB is depicted in Fig 1.5 and Fig. 1.6, with the positions of the individual components shown. These figures also show conductive traces and pads in black, drilled holes (u)(t) in gray and the silkscreen layers, containing logo and markings of components, in green on top and in blue on bottom side.

As it has been mentioned previously, the shield (PCB) is mounted onto the Arduino board using stackable header pins (p), that hold the two pieces firmly together while also providing electrical connections between all individual pins on the Arduino board, and their respective pins on the shield.

The purpose of the four smaller drilled holes (u) is to allow, by using hexagonal spacers, bolting the fan (c) directly onto the shield (b), and to prevent its unwanted movement. On the other hand, the purpose of the single bigger drilled hole (t) is to increase the amount of air that the fan can effectively take in.

Excluding the prices of postage, labor, external power adapter and the Arduino board, the total sum for of all components required to build a FloatShield R1 device is less than 30 €. Such an affordable price makes the device ideal for laboratories with limited budget and even for individual students.

Even with the other prices included, the total price remains moderate while all the components can be potentially reused in other similar devices such the ones proposed in [1–4].

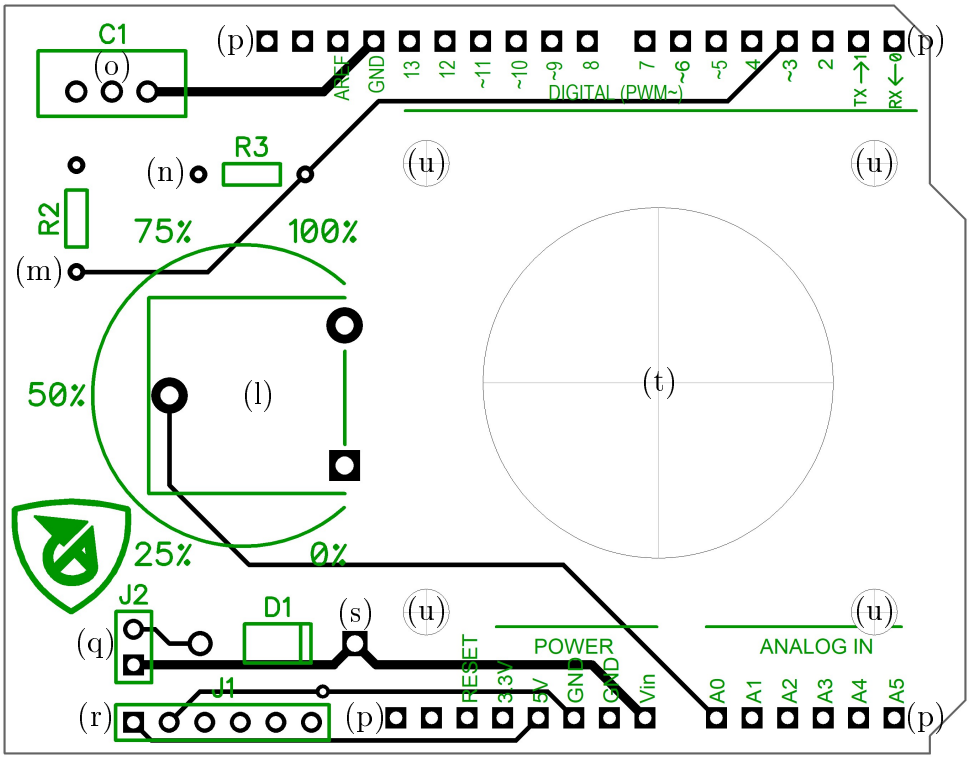


Figure 1.5: Top side of the printed circuit board of FloatShield R1.

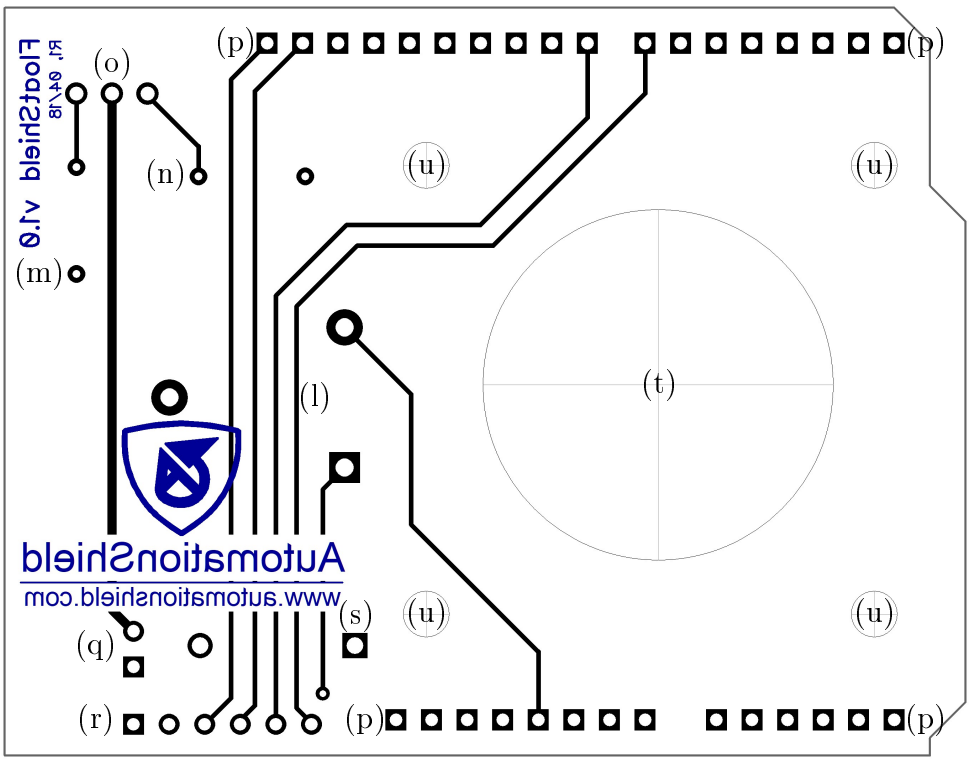


Figure 1.6: Bottom side of the printed circuit board of FloatShield R1.

1.4.2 R1 Software

The development, either of hardware or software, is a long and arduous journey. As it was previously mentioned, the creators of the first release of FloatShield were limited in time to one semester. Their journey was accompanied by plenty of trial and error, but in the end, they managed to create a fully functional prototype, and thus, lay a solid hardware foundation for this project.

In order to meet the conditions for successful completion of the course called Microprocessors and Microcomputers, they had to demonstrate to a jury of teachers that the designed and constructed device works, and that it can be actually used for its intended purpose.

Because most of their time was spent designing, manufacturing, and testing the hardware, there was simply no room to create an appropriate software support. Using the remaining time, they outlined a rough skeleton of the FloatShield library, and built a few basic functions needed to control the device to be able to demonstrate its fundamental functionality to the jury. Using the aforementioned functions, they created a sufficiently illustrative example of PID control that can be implemented on this device—thanks to which they successfully completed the course.

Their contribution to the FloatShield device has been considerable, and they shall be known as the ones who started it.

The rest of the thesis is built upon the original hardware.

Chapter 2

Hardware Design

As it is often the case with hardware, immediately after its creation, its imperfections and defects began to emerge—since it is unfortunately very difficult to predict some complications until they actually occur. But, that is what research, development and progress are about—one can make mistakes, however, one also learns from them, and in the end, comes out of the situation a little smarter. This was exactly the case with the FloatShield R1—from the very beginning, when the hardware was first assembled, various imperfections began to emerge. As it was already mentioned, these things are solved through releases within the AutomationShield project—where during the use of the device, its bugs are gradually collected and recorded with the intention that they will be corrected in the next release. Thanks to the fact that all information about errors and the device development is recorded online on the AutomationShield project website, it was possible to get acquainted with the current state of the device right from the start.

2.1 FloatShield R2

Among the first suggestions for enhancement in the next release was to: change the potentiometer to a smaller one, located further away from the fan for better access; move the fan and sensor connectors on the shield further apart to make them easier to disconnect; relocate one of the resistors (m), so that its legs protruding on the bottom side of the shield would not touch the conductive part of the Arduino USB connector and thus cause a short circuit.

More serious suggestions for enhancement that followed were: using a grid to laminarize the turbulent airflow in the tube; experimenting with the shape of the levitating object to improve its stability; exploring the libraries available for the TOF sensor; proposing necessary changes to the shield for ensuring compatibility with Arduino boards utilizing 3.3 V operating voltage in their MCU.

Both minor and serious flaws, were subject to scrutiny when creating the next version of the device. The first issue being dealt with, was turbulent flow in the tube.

2.1.1 Flow Laminarizing Insert

Figure 2.1 shows a CAD model of an experimental grid insert with a honeycomb cross-section. Its purpose is to force the air flow within the tube to become more laminar, what will eliminate unpredictable and chaotic movements of the floating object.

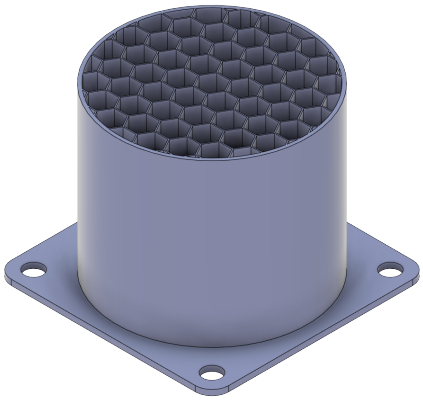


Figure 2.1: Experimental 3D printed honeycomb grid insert.

It was designed to be mounted directly atop of the fan, to be secured by the same screws that hold the fan, while the protruding grid would tightly fit into the bottom part of the tube. Such an arrangement secured, that the stream of air had no other option than to flow through the grid and thus become more laminar.

The idea behind it was solid and fairly straightforward, but the results were not satisfactory. While the air stream in fact became more laminar, what improved the behaviour of the ball in a sense that its undesirable oscillations have been significantly reduced, it produced another problem. The grid started acting as a throttle to the stream, what caused the fan to operate at much higher power output to be able to even lift the ball off the base (base in a sense of the lowest ball position in the tube). There were even cases, when at the start of an experiment, a power output of the fan was set to maximum, but the ball remained stationary on the base, and only after a significant period of time it begun to slowly rise to the top of the tube.

After careful consideration of the anticipated laminarity impact on the spherical object, it was concluded that the improvement brought by the grid insert did not match the objective. Therefore the honeycomb grid insert was no longer used.

A possible solution might be a new design of the grid with a different architecture of orifices, which would require a different approach. However this was not a central subject to this study.

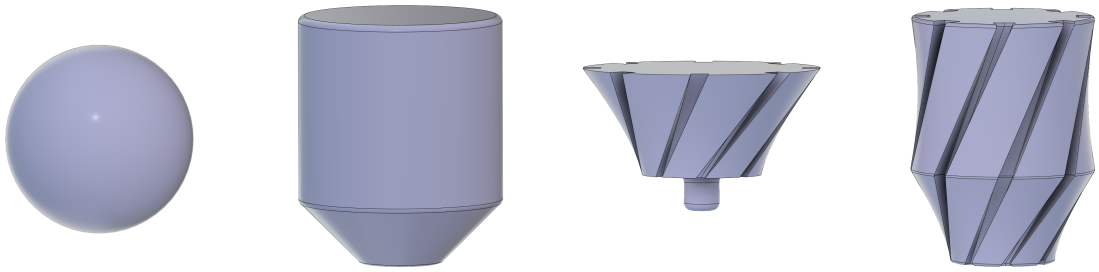
2.1.2 Floating Bodies

While a ball seems like a fairly solid choice of a floating object, it would be unreasonable not to try and see how different shapes will behave in the air stream.

As there are no commercially available items of required dimensions and material properties, it was necessary to create them using 3D printing.

For example, a diameter of an ideal floating object—a standard table tennis ball—is 40 mm, what makes it too large to fit into the tube currently used by the FloatShield device. Unfortunately, there are no other popular sports that use smaller ball sizes than that, with similarly low weight.

Figure 2.2 shows CAD models of bodies that were created using 3D printing, with the goal of improving upon the stability of ordinary ball shape and reducing unwanted oscillation.



(a) (b) (c) (d)

Figure 2.2: Experimental 3D printed floating bodies.

The solution to the "surprising" lack of suitable balls on the market was their 3D printing, what allowed creating almost perfect spherical shapes (2.2a) (only almost, as the sphere is a very difficult task for a 3D printer) of varying diameters. For a 15 mm object, the ball did not have a large enough surface to be even able to be lifted by the airflow. On the other hand, for 32 mm, there was only minimal space between the ball and the inner walls of the tube, what caused the ball not to float in the air, but rather it was being slowly pushed out of the tube—as only a little amount of air could get around the ball.

Another investigated shape was a cylinder with a conical bottom shown in Fig. 2.2b. The coned cylinder periodically wedged between the opposite walls of the tube during experiments, resulting in pulsation.

Next, the grooved designs shown in Fig. 2.2c and Fig. 2.2d have tried to use controlled rotation of the object, to avoid contact with the walls and thus possible wedging. Unfortunately, neither of them yielded the expected results. While the grooved cylinder (2.2d) was too heavy with regard to its small area exposed to the air flow and could not lift off of the base easily, the grooved spinning top (2.2c) had its center of mass located in its wider part, and due to that, it tended to overturn, so that the tip was facing up and the grooves could no longer fulfil their role.

In the end, it turned out that the laminarity of the air flow probably plays a much bigger role in the case of non-spherical objects.

3D printing is indisputably a technology of the future, but there still are certain bottlenecks preventing it from wide use, such as the variety of materials and time demand. Cork balls have been selected as a practical solution, available in shops providing model making supplies. These balls do not have a perfect shape, however, for the purposes of this project are satisfactory.

2.1.3 3.3 V Board Compatibility

The first of the two reasons why the FloatShield R1 is compatible only with boards utilising 5 V operating voltage is, that its potentiometer is directly connected to the `5V` pin on the Arduino board, and thus it outputs a signal with the amplitude up to 5 V to the `A0` analog pin. The operating voltage of an MCU is basically a logic voltage, the device requires to function properly and safely, and therefore it also approximately (as there is some tolerance) represents the highest voltage the input/output (I/O) pins of the Arduino board can safely handle without being damaged.

By mounting the FloatShield R1 onto 3.3 V board, even though it is physically possible, its analog signal reading pin `A0` would be in danger of being potentially damaged, and other parts along with it, as everything on the board is interconnected. A simple solution would be to set the potentiometer runner to its low output value and to simply not use it. However, that would be downgrading the device, rather than the actual solution.

A real solution could be introduced only in a new PCB, as the pathway between the potentiometer and `5V` pin had to be removed and, instead replaced with a pathway to the `3.3V` pin. That would secure compatibility with both types of Arduino boards—with 3.3 V boards as the potentiometer would now natively use 3.3 V as its maximum voltage output value, and also with 5 V boards as they could utilise the function of `AREF` pin that allows to set reference voltage for analog pins externally. This would require creating a pathway between `3.3V` pin and `AREF` pin, such that the reference would be externally set to 3.3 V and the potentiometer powered by 3.3 V would function properly even on 5 V boards.

2.1.4 New PCB

The compatibility-providing change could be realised only with modifying electrical pathways, what meant that a new PCB and its corresponding schematic had to be designed. The goal was to address as many flaws of the original R1 as possible.

Figure 2.3 depicts the new scheme, while the manufactured shield (PCB) is shown in Fig. 2.4. Note, that from now on, the parts that were changed or modified are marked with an asterisk (*), while the other parts remain unchanged and are the same as in the original release.

The most notable change in the second release of FloatShield R2 was the usage of a different fan (c*) that, in contrast with the previous one, is equipped with its own PCB that facilitates many new functions. For example, this fan (c*) contains its own controller for its power output, what means that the power output now could be regulated directly from the Arduino board by a PWM-capable pin, what made the MOSFET obsolete and thus, it was removed from the shield (b*).

Its other feature, as well as the main reason to choose this fan (c*), is the inclusion of a Hall sensor into its body, allowing it to directly output a signal to the Arduino (a). The signal is in the form of pulses, but with the knowledge, that each pulse represents one quarter of a full rotation of the fan blades, the rotations per minute (RPM) of the fan can be calculated.

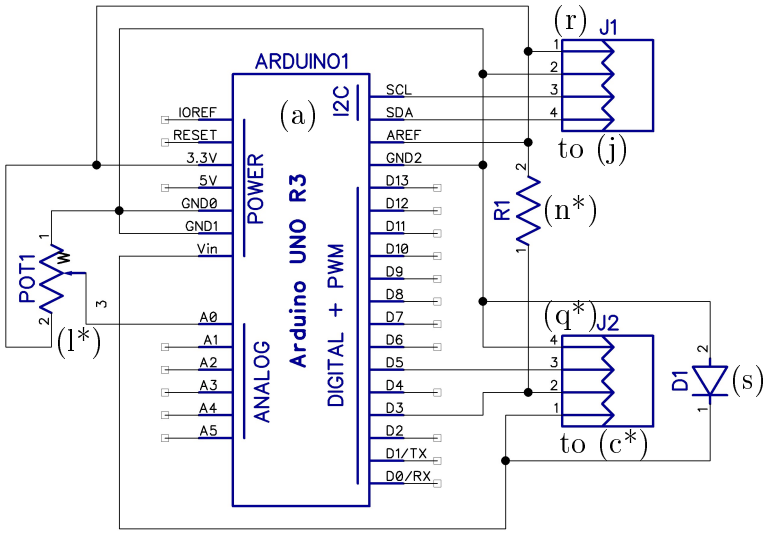


Figure 2.3: Electrical circuit schematic of the FloatShield R2.

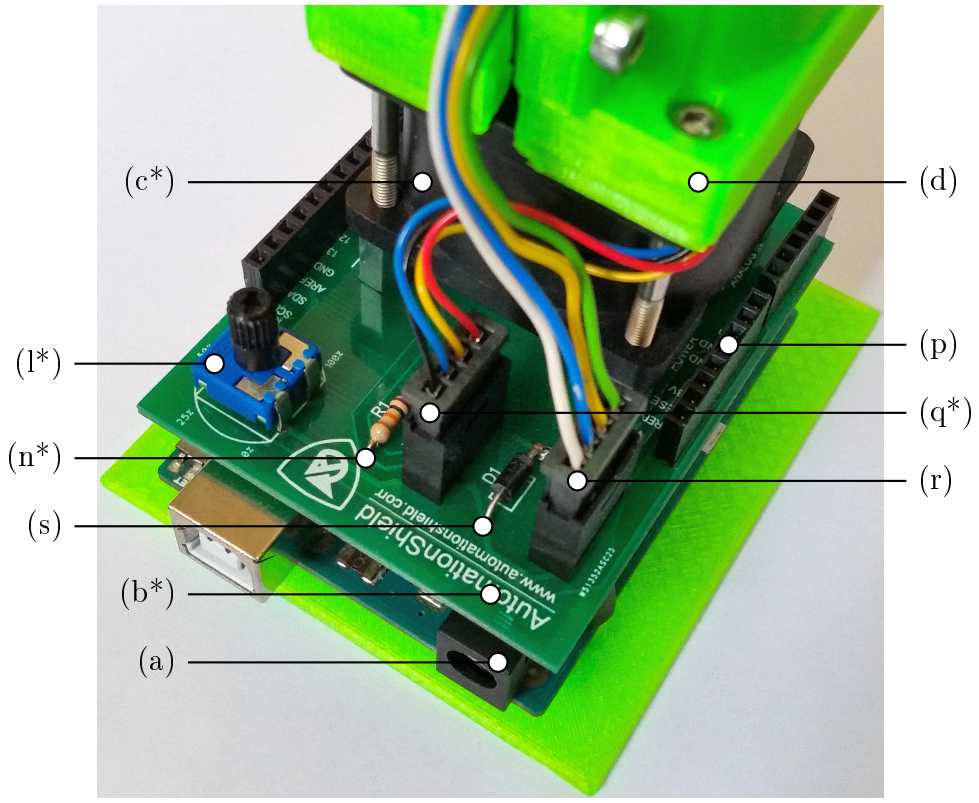


Figure 2.4: Detailed view of the lower part of the FloatShield R2 device.

Due to its new functionalities, the new fan (c*) was now connected to the shield (b*) through a connector (q*) that contained four cables, namely power, ground, signal from the Hall sensor and the PWM signal to the fan to control the power output.

A new 10 kΩ resistor was added to act as a pull-up (n*) for the Hall sensor functionality, while the two original resistors, that acted as a support for MOSFET have been removed.

The most notable of the more minor changes, are the usage of a smaller potentiometer (l*), that is more accessible for the hand of the user, and also changing the placement of the other components to make the shield (b*) more organized and transparent.

Figures 2.5 and 2.6 show the top and bottom side of the FloatShield R2 PCB, where as before, traces and pads are depicted in black, drilled holes (u),(t) in gray, and the silkscreen layers, containing logo and markings of components, in green on top and in blue on bottom. The new locations of all the individual components can be also clearly seen, including their corresponding connections to each other and to the board (b*).

These figures show another important change—that could be already seen on the schematic—namely, the implementation of compatibility with 3.3 V boards. Instead of using the `5V` pin, all the components of the shield (b*) are powered from the `3.3V` pin, notably the potentiometer (l*) and distance sensor ((j) through (r)). This change in itself enabled the FloatShield R2 to be used with 3.3 V boards, but only after also connecting the `AREF` pin to `3.3V` pin, the shield (b*) became fully compatible with 3.3 V and 5 V Arduino boards.

Another detail that can be seen more easily now is, that all three grounding pins `GND` are connected together. The idea behind this was, that although these pins are internally connected within the Arduino board itself (a), and act as a common reference ground, the fan draws relatively large amounts of current as part of its normal operation, what puts a strain on the paths as it goes through. As there is a possibility that the user might want to use one of those grounding pins for other purposes while operating the device, those extra pathways should act as a means of reducing strain on each individual path, and therefore protect them from a potential overload.

A direct connection between the fan (c*) and the `D3` pin of the Arduino board was possible thanks to the clever internal design of the new fan (c*), which despite being powered by the same 12 V wall adapter as the original one, outputs the signal from the Hall sensor by grounding the voltage level that is set by using a pull-up resistor (n*). This of course adds the need of a pull-up resistor, but on the other hand makes the fan much more universal, enabling user to choose the voltage level that is required by the MCU. In this case, the pull-up resistor pulls the reference up to 3.3 V by a direct connection to the `3.3V` pin, as the voltage of 3.3 V represents the logical high level on both 3.3 V and 5 V boards.

With all the aforementioned changes to the hardware, FloatShield R2 fully, or in some cases partly, addressed the imperfections of the FloatShield R1. Thus, a fully 3.3 V and 5 V compatible FloatShield R2 was born, with the added function of monitoring the real-time RPM of fan blades.

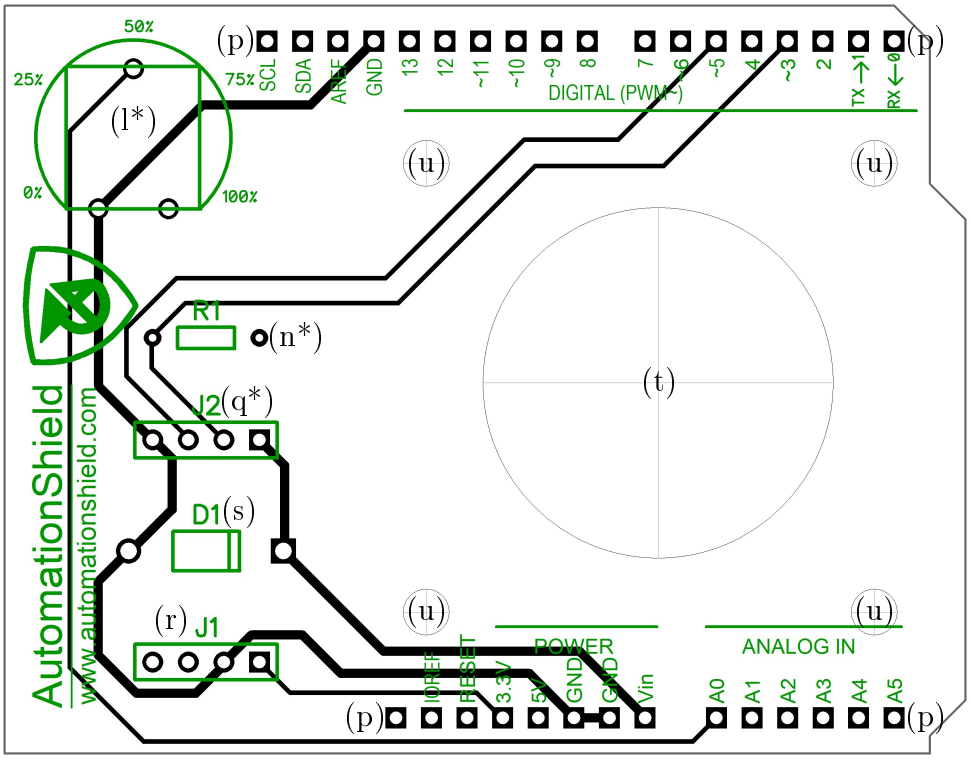


Figure 2.5: Top side of the printed circuit board of FloatShield R2.

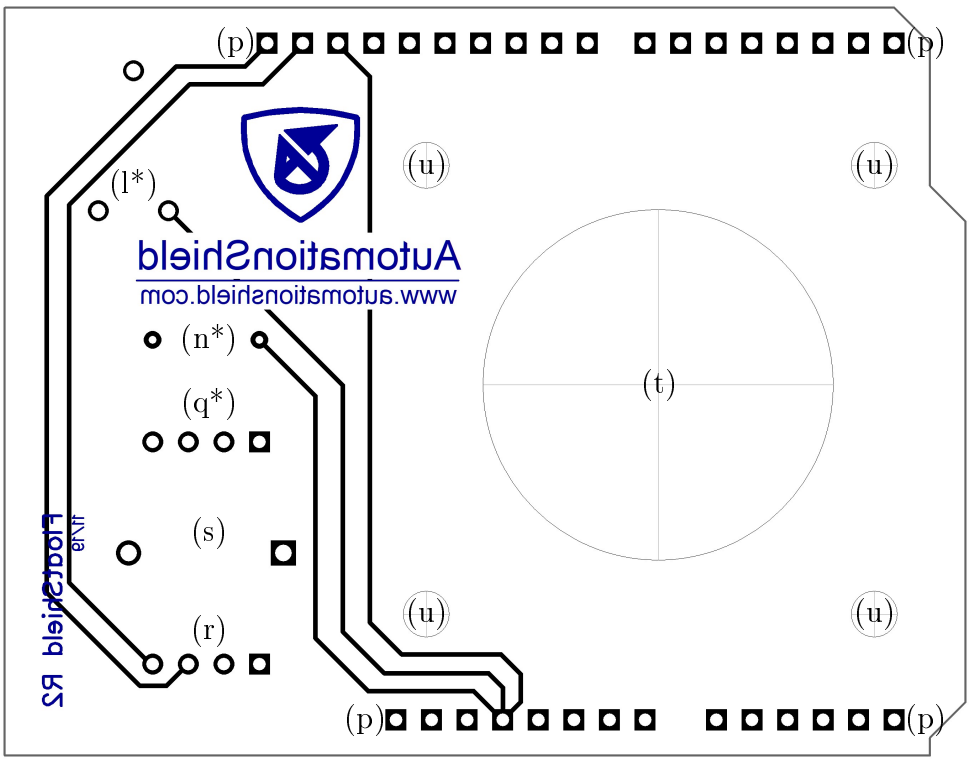


Figure 2.6: Bottom side of the printed circuit board of FloatShield R2.

2.1.5 New Sensor Holder

A new 3D printed sensor holder was designed at the final stages of R2 development. The CAD models of its two versions are shown in Fig. 2.7.

The main problem with the original holder, although it was nicely designed, was that it ironically did not hold the sensor properly. That is not to say that the sensor would tend to fall from the device, but the shape joint between the flange (h) and the holder (i) on the original device, was not firm enough. This practically meant that the holder, together with the sensor would slowly travel along the grooves in the flange (h). Those movements were not abrupt or significant, but they were there, and unwanted.

In the new design, the holder and the flange are one part, what effectively and firmly fixes the sensor directly onto the tube, and does not allow any unwanted movements. Note, that the sensor still needs to be screwed atop the holder the same way as before.

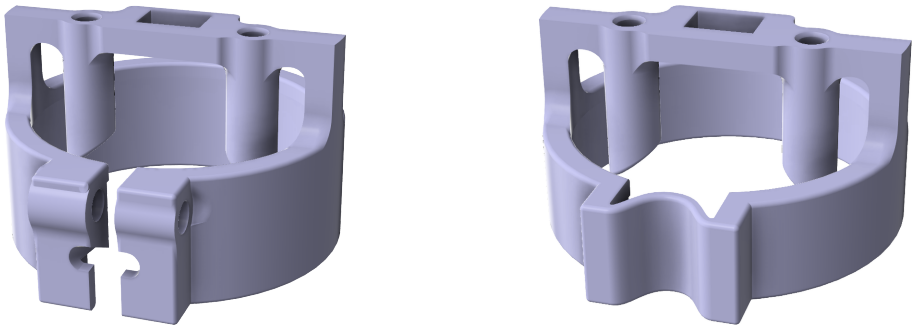


Figure 2.7: Two variants of experimental 3D printed sensor holders.

There are two variants, one that is fixed onto the tube using a bolt and a nut (left) and other, that is created with slightly smaller diameter than the tube and therefore once mounted, is held using frictional forces (right).

A little design detail is, that the two thin cylinders, whose main purpose is to act as holes for the screws and at the same time as a distancing bodies that protect the sensor from the ball impacts, have chamfered ends—to better accommodate the spherical shape of the ball.

2.2 FloatShield R3

As the beginning of this chapter states, the nature of development is often based on trial and error, accompanied by learning and thus continuous enhancement. Unfortunately (or rather fortunately?) there was something for us to be learned from the FloatShield R2.

The major flaw of the FloatShield R2 device, despite it being fully compatible with all types of Arduino boards with R3 pinout, and despite providing additional RPM measurement of the fan blades is, that the internal controller within the fan regulating fan power output introduces a certain delay into the whole control chain. The fan reacts to the change introduced in the duty cycle of the input PWM signal, only after a few seconds of initial inactivity.

This behaviour is highly detrimental in such a delicate system, where the ball position is sampled with time periods ranging in tens of milliseconds, and it is expected of the fan to be able to react as quickly as possible.

Lesson to be learned is, that it is not a good idea to use such an overly complex fan, whose original purpose was to probably act as cooling fan of a processor heat sink, where the few second difference does not matter. Especially in a system, that relies on fast reactions of the actuator.

With the knowledge that a "four wire" fan with its own unknown internal control—that would pose even greater problems for system identification—can not be effectively used in the FloatShield device, the direction of the next release should be reverse. Specifically, to the simpler "two wire" fan that on one hand requires additional components like MOSFET, but on the other, provides an almost instant reaction to the input change.

Based on the aforementioned findings, a possible FloatShield R3 can be proposed. Its schematic is shown in Fig. 2.8.

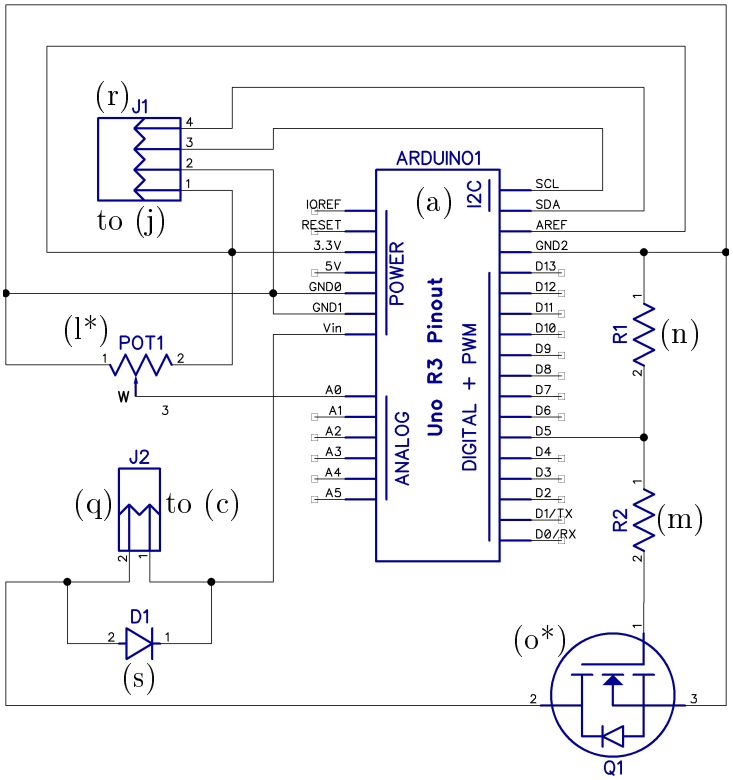


Figure 2.8: Proposed electrical circuit schematic of the FloatShield R3.

As it can be seen in Fig. 2.8, the proposed FloatShield R3 would be a combination of the original FloatShield R1 and its successor FloatShield R2—it would utilise exactly the same components as R1 with the only two exceptions being the smaller potentiometer (l*) introduced in R2 and a different MOSFET (o*). At the same time it shall be powered from `3.3V` pin, interconnect `GND` pins, and also bring the 3.3 V to the `AREF` pin, as it was done on R2. That should effectively combine the best parts of both releases, while leaving out the worst.

The first of the two reasons why the original FloatShield R1 was compatible only with boards utilising 5 V operating voltage has been explained in the Chap. 2.1.3. The second reason was found out only recently, while trying to temporarily patch the FloatShield R1 to allow its use with 3.3 V boards.

As it was also mentioned in Chap. 1.4.1, the MOSFET transistor acts as a switch that does or does not allow a current to flow through the circuit, based on whether sufficient voltage is present or not present between its `GATE` and `SOURCE` pins. The "Gate-Source Threshold Voltage" parameter found in datasheets, represents the typical voltage required to properly "open" the `GATE` and, it often has a range listed—as the datasheet has to take into account that despite all components being manufactured theoretically the same way, they are not in fact the same. For the specific MOSFET used in the FloatShield R1, this range is minimally 2 V and maximally 4 V, what can be interpreted that the used MOSFET will need at least 4 V on its `GATE` pin to allow proper current flow, anything below might restrict the flow partly or completely.

At this point, it is obvious why there would be need for a different MOSFET—the one used in R1 can be utilised only with 5 V boards, as only they can provide sufficient voltage from their pins to fully open the MOSFET `GATE`, while the 3.3 V can not. This fact was confirmed on the device, where the R1 was powered from the `3.3V` pin and connected to the 3.3 V board. Everything worked as it should, except the fan. Voltage was measured by a voltmeter on the `GATE` pin, coming to 3.2 V, and then 5 V was brought from another board directly onto the `GATE` pin—what caused the fan to turn on.

Figures 2.9 and 2.10 depict the proposed design of the PCB for the FloatShield R3 device. Theoretically, the proposed FloatShield R3 device should combine all the enhancements brought by FloatShield R2 with the dependability of FloatShield R1. The only condition being, that the gate threshold voltage of the used MOSFET will be equal or ideally slightly lower than 3.3 V.

In the end, the FloatShield R2 turned out to be practically unusable for any serious experiments because of the delayed reaction time of its fan, that was originally meant to be its greatest strength. Due to that, only the FloatShield R1 device compatible with 5 V boards was used throughout the following chapters.

The schematics and PCB design of the possibly best version of the FloatShield—FloatShield R3—are proposed here with the hope that they will inspire the real third release of the device, as it was not possible to create the FloatShield R3 hardware at this point in time because of the ongoing SARS-CoV-2 pandemic.

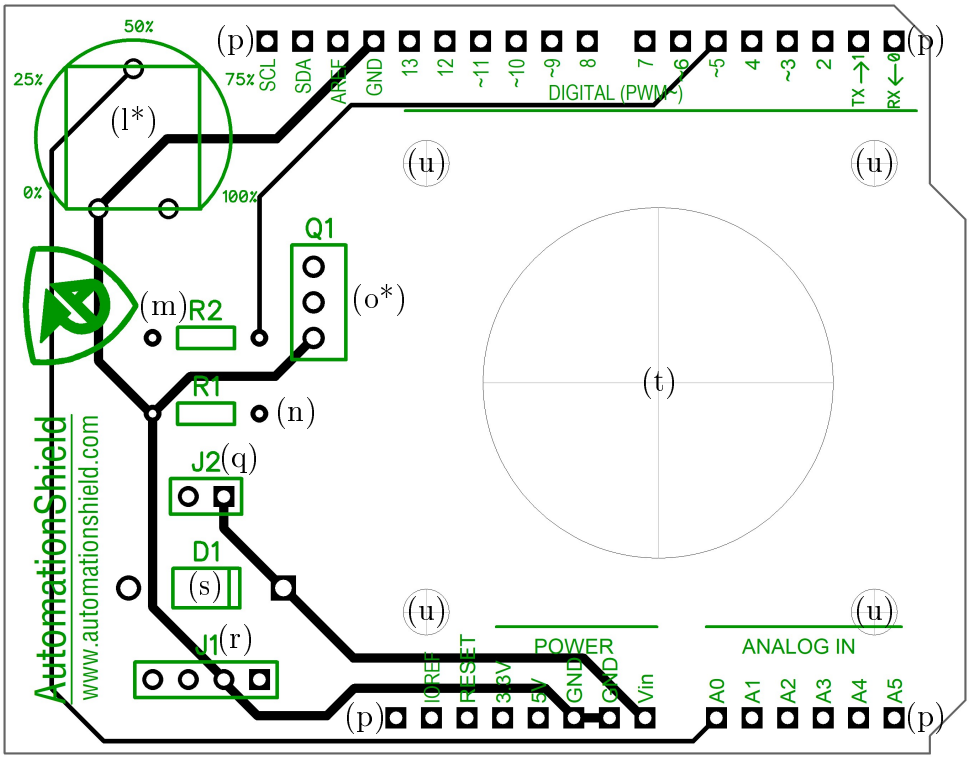


Figure 2.9: Top side of the proposed printed circuit board of FloatShield R3.

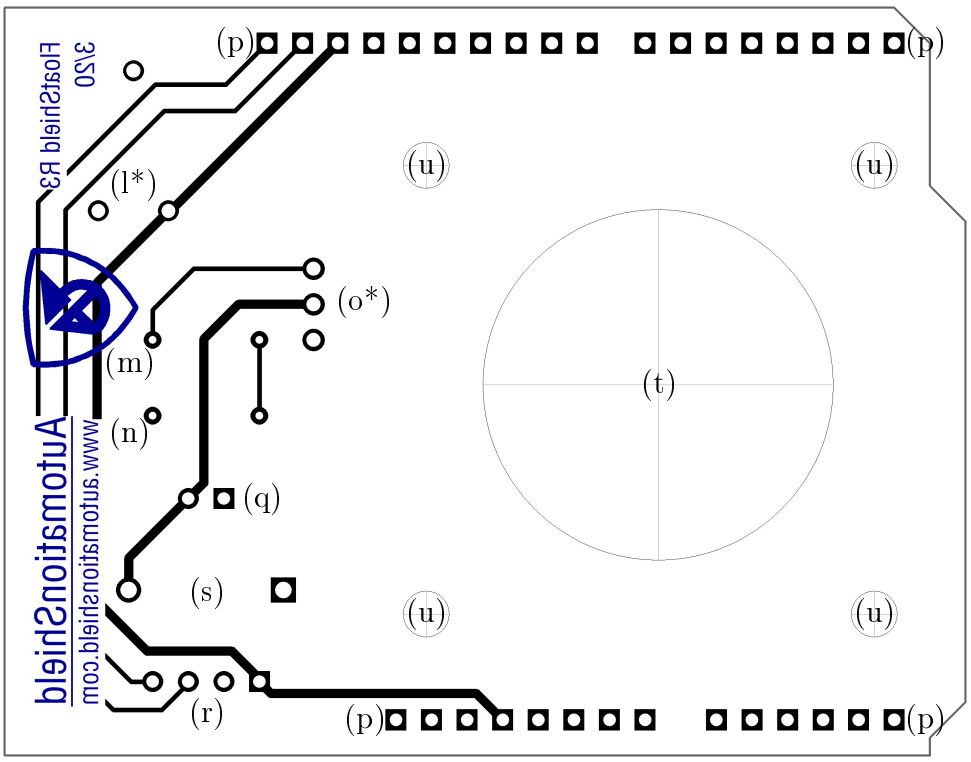


Figure 2.10: Bottom side of the proposed printed circuit board of FloatShield R3.

Chapter 3

FloatShield API

The open-source software support for the FloatShield device is realised through an application programming interface (API) that, with numerous functions included within it, provides the user with a multitude of useful universal tools and makes the operation of the FloatShield device significantly easier and beginner friendly.

As the FloatShield device is part of a wider effort of creating open-source educational aids for control and mechatronics teaching, the API for FloatShield and all other shields under the wing of AutomationShield are included within the "AutomationShield" library. All the user needs to do is to connect the device and include the AutomationShield library into the selected software environment. That enables one to deploy the predefined functions for each and any of the AutomationShield devices.

3.1 Supported Software Environments

Currently there are three different software platforms supported by the AutomationShield devices—meaning that there are three AutomationShield libraries or rather three environment-specific versions of the AutomationShield library. The individual versions of the library contain mostly the same functions when it comes to their purpose, with the only difference being their implementation—as the various programming languages have different rules.

The software platforms currently supported by the AutomationShield are:

- Arduino IDE
- MATLAB
- Simulink

It is important to note that, while not necessarily all of the AutomationShield modules currently support all three software platforms, all of them do support at least one of them (specifically the Arduino IDE), and most of them support at least two. The goal is, of course, a complete support of all the shields, but it is still a work in progress—each shield progressing with its own pace.

Fortunately for the purposes of this thesis (and as the result of this work), the FloatShield device is as of now being supported by all of the three aforementioned environments.

In each of the AutomationShield library versions, there have to be that version specific functions for each device, what can also be interpreted in a way that there are three versions of libraries for each shield, which are then included in the major AutomationShield library. This interpretation is actually very close to the practice of how AutomationShield handles the functions of individual devices, since each device indeed has its own library, however, at the same time, each of those libraries is dependent on the main library containing many universal functions. Together, they create an interdependent system, called the AutomationShield library.

Based on this, the support of the FloatShield device is implemented through three environment-specific libraries.

3.2 AutomationShield Function Naming Conventions

One of the many efforts of the AutomationShield project is, when it comes to software, keeping the names of its functions and key variables simple, descriptive and consistent across all of the supported software platforms for all of the developed shields. It would be very ineffective and confusing if the user had to learn the names of even the most basic functions from scratch, when either just going from one software environment to another, or when trying a different AutomationShield device.

A guide for the AutomationShield developers and students under the name "AutomationShield API Style Guide" has been created for this purpose, where among other things, the recommended names for the functions with similar purposes are proposed. Thanks to that, regardless of the used device or the software environment, there will be applied similar, if not exactly the same names of the functions.

Let us mention a few examples of the basic functions—to initialise and otherwise prepare the device for operation—the user will have to use `begin()` function. It is generally required to first run the `begin()` function in order to use all the other functions. All of the standard tasks that have to be performed on the device only once at startup, can be found within it.

In most cases, after starting the device, the next step is to perform a calibration of the sensors—to acquire certain reference values that are necessary for other functions. This step is often required to make sure that the following functions will return accurate values, since the individual units of the same device have slight differences, and a single constant can not accurately describe them all. The function logically named `calibrate()` should be used for this purpose.

After starting the device and getting all the important values from calibration, the next stem is to use it—mostly by acquiring values from its sensors, processing and interpreting them and then, possibly, deciding how the actuators should act.

The measurement acquisition part is then realised through the `sensorRead()` function, which should return the values measured by the sensor on the device in some usable form, while the actuation part is realised by the `actuatorWrite()` function, that should universally expect an input argument representing the amount of effort the actuator has to produce—expressed in percentage.

The last function worth mentioning, as it is used with all the AutomationShield devices that are equipped with a potentiometer runner (most of them are), is meant to provide

the user with some form of additional influence over the course of the device operation. The function `referenceRead()` thus returns the value gained from the position of the potentiometer runner, often further mapped to a more practical scale, for example, into percentages. It can then be used to provide a manually changing reference value to some ongoing process on the device.

These aforementioned functions should be universal, and should serve similar purposes on all the devices in all the supported software environments. Then, if in the following chapters, a function with the same name and structure is mentioned for each software platform, it is not a mistake, quite the opposite—the naming convention is just being implemented correctly.

3.3 FloatShield API for Arduino IDE

3.3.1 About the Arduino IDE

The main reason that the Arduino IDE is supported by all of the AutomationShield devices is, that it is an open-source integrated development environment provided and maintained by the Arduino company itself—the manufacturers of the original Arduino boards. The fact that the programming environment was built by the same people that manufacture the hardware, provides a certain amount of credibility and reliability, and as it is free to download, it became the default IDE for most Arduino users.

In addition to being beginner friendly and full of directly uploadable examples, the Arduino IDE includes many useful predefined functions and constants. These greatly help the user with creating specific programs and therefore operating the Arduino device.

Thus, there was no reason for AutomationShield to choose any different environment for the Arduino-dependable educational devices.

The programming language used in the Arduino IDE software environment is the C++ language and because of that, it would be ideal if the user knew at least the basics of this language before starting to use Arduino and AutomationShield devices.

But even if that was not the case, thanks to the immense popularity of Arduino platform, the internet is full of comprehensive tutorials for beginners, that guide them through various projects, and in the process, teach them the basics of programming in the C++ language.

All functions in the AutomationShield library are logically named and described in detail, what makes them very friendly even for the beginning Arduino users that wish to learn more about the nuances of the C++ programming language.

Any user of the Arduino IDE can download the freely available AutomationShield library from the project website, and then take the advantage of the friendly graphical user interface (GUI) of the Arduino IDE, that allows to search for the library through the files on the computer and include it into the so-called sketch file.

The "sketch" file with the extension .ino is the default source code format for the Arduino IDE.

When the user is ready to test the written code, it can be compiled and then uploaded into the specified Arduino board directly using the GUI of Arduino IDE, through a USB cable connected to the USB plug of the board.

3.3.2 FloatShield General Methods

As it is common in C++ and many other programming languages, the functions are stored in so-called libraries. In the case of Arduino IDE, these libraries sometimes consist of only two files, where first is the header file (.h suffix) that contains function declarations—which describe the names of the functions and information about their input and possible output parameters—and it also contains macro definitions that most often represent important constants.

The second file is the source file (.cpp suffix) that contains definitions of the functions declared in the header file—these describe the specifics of how the objective of the function is actually implemented.

Together, they can serve as a very simple library of functions in the Arduino IDE.

Note, that generally the term "library" represents a large number of interdependent header files and source files—not just a single pair.

What is more specific to the C++ language is, the inclusion of the object-oriented programming (OOP), and the AutomationShield libraries take advantage of that feature. In reality, this means that all of the functions "belong" to a class, where the class is a type of structure that allows to describe real life systems and concepts with their properties and then represent them as single a variable (called object). Therefore, each of the shields has its own class that tries to faithfully describe all of its attributes in the form of various variables and constants. An object created from this class can be then thought of as a representation of an actual physical device within the program.

This is done within the header file, where all of the functions are declared as a part of the class representing the specific system.

In the case of FloatShield, the class representing its properties is named `FloatClass` while the object automatically created by default for the user at the end of the header file has the name `FloatShield`.

Now all that needs to be done to be able to use the functions for the FloatShield device, is to include the FloatShield API header file `FloatShield.h` through the preprocessor directive `#include` at the beginning of the Arduino IDE sketch file. As an instance of the `FloatClass` is already created by default for the user at the end of the header file, the methods of the `FloatShield` object can be used right after including the header file into the sketch.

The code of the FloatShield Arduino IDE API can be found in appendix A. Specifically, the content of the header file `FloatShield.h` is provided in A.1, and the content of the source file `FloatShield.cpp` is provided in A.2.

The FloatShield library is included at the beginning of the Arduino IDE sketch file by using

```
#include <FloatShield.h>
```

After that, all the functions contained within the `FloatShield.h` header file become available to use.

As they are now part of a `FloatShield` object, they can be called only through that object by utilizing the dot operator, since in the C++ the dot operator (.) is used to

access the individual members of the class object be it a variable or a method belonging to that specific class.

Note, the function belonging to a class is in the object-oriented programming (OOP) terminology called a method.

The C++ classes are a practical way managing all of the important variables and functions and keeping them at one place.

According to the aforementioned logic, the FloatShield device can be started after including the header file by calling the

```
FloatShield.begin();
```

method of the `FloatShield` object. This method launches the I2C communication interface between the Arduino board and the TOF distance sensor and initialises the sensor—confirming that the connection has been successful, and if it was, puts the sensor into high speed ranging profile by setting its ranging timing budget to its lowest value—20 ms. In this mode, the sampling speed is prioritized over accuracy, as in the case of the FloatShield device, it is more important to react quickly than to get the most accurate measurement.

After setting up the sensor, the method starts the measuring in continuous mode, where the sensor automatically and periodically (in this case every 20 ms) measures and stores the distance, and the user only needs to access this value whenever it is needed.

The last thing performed by this method is initialising certain working parameters of the `FloatShield` object, for example, a flag carrying the calibration status of the device is set to `false` as the device was not calibrated yet. Moreover, the minimal and maximal values of the distance measured by the sensor—that are normally being found during the calibration process—are set to their approximate values. This is done for the case that the user would, for some reason, not want to calibrate the device before using it, but wanted to use its other functions—as certain functions do rely on those parameters.

The calibration procedure is then called by the method

```
FloatShield.calibrate();
```

Its purpose is to experimentally find the minimal and maximal values of the distance measured by the TOF sensor. That is accomplished by setting the power output of the fan to its maximum, and while the ball is at the very top of the tube, measuring a certain number of samples acquired from the distance sensor. After that, the process is repeated with the power output set to zero and the ball lying on the base.

With a known number of samples that were taken in both positions, the measurements from the upper part of the tube are averaged, and the result is saved to the `FloatShield` object that stores a minimal value measured by the sensor. The measurements from the lower part of the tube are averaged the same way, and the result is stored into the `FloatShield` object that stores a maximal value measured by the sensor.

Note the inverted relationship—as the sensor is mounted at the top, the ball being in the upper part of the tube means, that the distance between the sensor and the ball is

smaller than in the case of the ball being in the lower part of the tube, where the distance between them is greater.

The approach of averaging a certain number of measurements to get the final value was chosen instead of simply taking the greatest and smallest values of respective measurements, because the sensor in high speed mode sacrifices the measurement accuracy for higher sampling speed, what in reality means frequent small deviations and generally more noise in the measurements. By assuming that the deviations are of similar magnitude to both directions (closer to and further from the sensor), the averaging approach seemed to be more faithful to reality.

After acquiring the (average) extreme values of the TOF sensor measurements, the working distance range of the sensor is calculated by subtraction, and this value is then stored in the `FloatShield` object.

When done with the calibration process, the method sets the flag carrying the calibration status of the device to `true`, as the calibration was successfully executed.

This informative variable can then be accessed by the method

```
bool calibrationStatus = FloatShield.wasCalibrated();
```

which returns a boolean (`true` or `false`) value representing, whether the device has been already calibrated somewhere within the sketch before.

Note, that the boolean variable `calibrationStatus` is created here only as an example of storing the return value of the `wasCalibrated()` method. A similar illustrative variables will be also created with the following methods of the `FloatShield` object.

The values acquired during the process of calibration—the extreme values measured by the sensor and the calculated range (all of them expressed in millimetres by default)—can be accessed by using

```
float minimalDistance = FloatShield.returnMinDistance();

float maximalDistance = FloatShield.returnMaxDistance();

float measurementRange = FloatShield.returnRange();
```

Additionally, the raw measurement of the distance expressed in millimetres from the TOF can be accessed by the method

```
float rawDistance = FloatShield.sensorReadDistance();
```

Nevertheless, directly using this raw value of the distance in millimetres from the sensor mounted at the top of the tube would be impractical, as measuring the distance from the top is very unintuitive. The smaller reading means that the ball is higher while the bigger reading means that the ball is lower in the tube, while it is more natural to interpret the higher value as a higher position and lower value as a lower position.

Note, that this method, and the following methods concerning the acquisition of measurement, can be called at most every 20 ms, as that is the physical limit of the sensor

set by its manufacturer. Calling these methods more often than that would result in repetitive measurements as the sensor actualises the measured value in its registers only once in every 20 ms.

The problem of unintuitive position readings is solved by the method

```
float ballAltitude = FloatShield.sensorReadAltitude();
```

that, by using the values acquired during the calibration process, inverts the reading. Instead of raw distance data from the sensor, this method outputs the altitude of the ball—its distance in millimetres from the base.

On the other hand, the method

```
float percentualAltitude = FloatShield.sensorRead();
```

goes one step further, which additionally to inverting the reading to measure the distance from the bottom of the tube, also maps the reading from its original range to the percentual range by using the values acquired during the calibration process. Its output is then an inverted and rescaled reading, that represents the current altitude of the ball expressed in percentage—0% being the bottom and 100% being at the very top of the tube.

This percentual measurement scaling becomes useful in the case of displaying the ball position together with the actual power level of the fan in the *Serial Plotter* tool provided within the Arduino IDE, where the real-time plots are scaled to the same axis range. As it is only natural to express the power level of the fan in percentages, expressing the ball position in percentual range allows to display these two values together on the same plot, without compromising visibility of either of them.

To control the power output of the fan the user can call the

```
FloatShield.actuatorWrite(u);
```

method, that expects input `u` in the percentual range—from 0% up to 100%—based on which it regulates the power level of the electric motor in the fan.

How that is performed exactly has been mentioned in Chap. 1.4.1 when explaining the reasons of using the MOSFET transistor in the FloatShield R1. To summarise it, the fan has only two connectors, and therefore when it comes to its use with Arduino boards, it can be only turned on or turned off—it does not have any native means of regulating its power level. To devise such means, the MOSFET transistor was introduced, taking advantage of the PWM functionality of specific pins on the Arduino boards. By connecting the pin capable of creating the PWM signal directly to the `GATE` pin of the MOSFET, the MOSFET became a switch capable of turning the fan off and back on in quick succession. The Arduino board can control the duty cycle of its output PWM signal, and therefore, by controlling the duty cycle of the PWM signal sent to the `GATE` pin, it is possible to effectively control for how long will the fan be turned on within a set period of time, and thus regulate its power output.

The input `u` to this method can then be simply thought of as duty cycle of the fan itself, or rather directly, the power level of the fan.

In case the user mistakenly inputs a value out of the expected percentual range, the method first constrains the input to match the expected limits.

After that, the percentages are mapped onto a 8-bit scale of integer values from 0 to 255, as the specific timer connected to the pin is only capable of creating a PWM signal with 8-bit resolution. This integer value is finally sent to the Arduino board, that modifies the shape of its output PWM signal accordingly.

To utilise the functionality of the potentiometer built into the FloatShield, the user should call

```
float percentualReference = FloatShield.referenceRead();
```

method, that outputs the current position of the potentiometer expressed as a floating point number in a percentual range 0%–100%.

The function takes the reading from the pin with ADC functionality, into which the wiper of the potentiometer is connected, and then it maps this reading from its 10-bit value—from the range 0–1023 (as the ADC on the Arduino board has a resolution of 10-bits)—to the percentual range 0%–100%, and then returns the result.

The second variant of this potentiometer position reading method, that is better suited for more specific purposes can be called by

```
float refAltitude = FloatShield.referenceReadAltitude();
```

method, which in contrast to its previously mentioned variant, instead of mapping the raw ADC reading to the percentual range, maps the raw reading to the altitude range of the ball in millimetres, by using the values acquired during the calibration process. That means that one end of the potentiometer wiper position represents the zero altitude while the other end represents the altitude of the ball when being at the very top of the tube.

This variant is more convenient for the cases when the user reads the ball altitude in millimetres by using the `sensorReadAltitude()` method, and wants to compare or possibly control it to some reference value. In such a case it only makes sense that the reference would be in the same range, and the same units as the observed position value.

All of these aforementioned methods are universal in a sense, that every release of the FloatShield device can use them, and together they represent the core of the FloatShield library for the Arduino IDE platform. The code of the individual methods can be found in appendix A.

3.3.3 FloatShield Release Specific Methods

In addition to the base methods, there are a few release specific methods that can be used only on a particular release of the device, as they are tied with its additional functionalities.

Currently, there is only one another release besides the original—and that is the FloatShield R2. It has two main dedicated methods based on its additional RPM reading capabilities.

If the user has a FloatShield R2 device, all that needs to be done to take advantage of its additional functionalities, is to change the preprocessor token defined in the `FloatShield.h` header file (provided in A.1). This token specifies, which release of the device is being used. Specifically, it would be changing the preprocessor token definition `#define SHIELDRELEASE 1` into `#define SHIELDRELEASE 2`, what would give the compiler all the information it needs. Based on this choice, all important constants (or preprocessor tokens) are defined, representing for example, pins used by the specific release. Also, by utilizing conditional compilation, it is determined, based on the value of this token, which functions will be available to the user.

Therefore, if the user owns a FloatShield R2 device, and has successfully specified this within the header file of the FloatShield library, the first of the two main methods, can be accessed by

```
float fanRPM = FloatShield.sensorReadRPM();
```

which returns the actual angular velocity of the fan blades in RPM. It interprets the signal coming from the Hall sensor built-in to the fan as the RPM value, and it does that by utilising external interrupt functionalities of certain pins of the Arduino board. After reading the signal from the Hall sensor, with the pin capable of interrupting the processor based on external signals, it is possible, to count the number of pulses that occur in a set period of time very precisely.

Then, by correctly interpreting what the pulses represent (this information is present in the fan datasheet), and by counting how many pulses did occur in a set period of time, the RPM can be calculated and returned.

While its design is quite clever, the accuracy of the method depends on whether the time between its individual calls in the loop is known—the code within the loop is being periodically sampled with a known sampling period. The set time period is then figured out from the sampling period.

The sampling period is set to 25 ms by default—the function assumes that it is being called approximately at every 25 ms, and based on this assumption, it calculates the RPM.

In case the previous assumption is false, it is advised to utilize the

```
FloatShield.setSamplingPeriod(Ts);
```

which can be thought of as a support method for `sensorReadRPM()`. It expects an argument `Ts`, that represents the sampling period in milliseconds that is being used to sample the loop where the RPM are observed with the `sensorReadRPM()` method.

The only purpose of this function is, to specify the sampling time if needed—to ensure that the `sensorReadRPM()` method for calculating RPM will return accurate results.

The second of the two main methods specific for the second release of the FloatShield device can be called by typing

```
FloatShield.actuatorWriteRPM(RPM);
```

within the sketch. The method expects one input argument—a value of the `RPM` that the user wants the fan to reach.

This functionality, of setting the fan power output by specifying the `RPM` was possible thanks to the ability of measuring the `RPM` with the `sensorReadRPM()` method. The `sensorReadRPM()` method was used, to find the relationship between the power levels of the fan and their corresponding `RPM`. It turned out to be a simple linear relationship and, by fitting a line over the measured values, it was possible to find its corresponding equation.

The method accepts input in the range from 0 up to approximately 17 000 min^{-1}, and based on the known relationship between the power level percentages and `RPM`, sets the power level of the fan to the corresponding value. Its accuracy can be confirmed by using the `sensorReadRPM()` method and observing if the set `RPM` are the same as the measured `RPM`.

3.3.4 FloatShield Special Methods

During the work on the FloatShield R1 device, certain specialised functions had to be created, to serve the particular purposes of more complex examples.

Specifically, these concerned performing matrix operations, that the Arduino IDE on its own is not effectively capable of. However, as in most cases with the Arduino platform, if the functionality sought by the user is not built into the Arduino IDE directly, there surely is an external library that can provide it.

The requirements on such a library were: that it shall be able to conveniently store matrices; to enable the user to perform essential matrix operations, such as, addition, multiplication or inversion; and to perform these operations efficiently—in the least amount of time.

A research was conducted on this very specific matter in [18], where the author provides comprehensive explanations and comparisons of individual available libraries by actually applying them in complex matrix algorithms. Based on the results of particular libraries presented in [18], the Basic Linear Algebra (BLA) library was chosen to serve as a tool to perform matrix operations within the Arduino IDE FloatShield library.

The BLA library enables the user to create matrices by specifying their characteristic dimensions—number of rows and number of columns—and then entering the values of their individual elements.

This need of specifying the dimensions, could prove to be impractical in cases such as a function taking these matrices as its input arguments. The function would need to know their dimensions to be able to work with such matrices—they would have to be provided as additional input parameters. It might not be a problem for smaller functions that would operate with a few matrices, but it would make the function calls of more complex functions very chaotic.

To be able to create more complex BLA matrix functions, the so-called templates were used. Templates are another very useful tool within the C++ programming language (along with classes), as they allow the creation of functions (among other things) with unspecified types of their input and output parameters. That does not mean that the arguments provided will not have any type—as all created variables simply have to have a type explicitly defined. What that means, is that the type of their parameters is explicitly

declared during the function call by the user. Based on that, the functions know what types to expect as an inputs, and what types to return. Then, there does not have to be the same function created for each different data type—one template function can serve them all.

That is, however, not exactly our case, as the types of input arguments that will be provided to, and output arguments that will be returned by the function, are known—they have to be BLA matrices. Thankfully, the templates can be implemented in different ways than for the aforementioned function type-defining purpose. In our case, the problem is figuring out the dimensions of the matrices provided as input arguments to the function. The template functionality which will be used to achieve this, is the so-called template argument deduction.

Template argument deduction, as its name suggests, can deduce certain parameters of the input arguments. For example, in our case, the sought parameters are the dimensions of a matrix that will be provided to the function as an argument, while it is certain that the dimensions will be represented with two numbers of an integer type. What has to be done, is to define two template variables of integer type, which will represent the dimensions of the provided matrix. These will be used inside the template function, as expected dimensions of the input BLA matrix. The deduction part happens during compilation, where the template variables are replaced with actual integer dimensions of the matrices, and the function works the same way as if the user would provide all the dimensions explicitly.

Then, if such template function is designed correctly, it can take in matrices of arbitrary dimensions, execute the specified operations, and return matrices of arbitrary dimensions, without ever being explicitly provided their actual size. This becomes an effective way of creating complex functions that operate with matrices.

There are currently two special template functions, and as they have been incorporated into the FloatShield library and `FloatClass` class, they have also become its methods.

The first of the two special template functions was created to suit the needs of experiments concerning the LQR system control in a closed loop. Its specific code is provided in appendix A.3. The user can call this method by

```
BLA::Matrix<rows,columns> K = getGainLQ(A,B,Q,R);
```

where all the arguments `K`,`A`,`B`,`Q`,`R` are `Matrix` objects of the `BLA` class, and their respective dimensions—`rows` and `columns`—have been set during their definition. In the case of `K` being defined here, they have to be provided directly—the user has to know what dimensions it will have, based on the input matrices provided to the function.

These arguments represent the matrices commonly present in LQR control examples. Specifically, the matrices `A` and `B` represent the state-space matrices of the controlled linear or linearised system—they store the information about its behaviour and its reactiveness to the input. On the other hand the matrices `Q` and `R` represent the penalisation matrices of the system states and input respectively—they store the information about the priority of the respective system states with which should the LQR try to control them to their reference values (`Q`), and the information about how aggressively should the system try to achieve that control (`R`).

With these four matrices known (`A`,`B`,`Q`,`R`), it is possible to attempt solving the infamous Riccati equation, and as a result to acquire the stabilizing gain `K`—the LQR gain.

The specific approach used to solve the discrete Riccati equation has been inspired by [18], where an iterative algorithm under the name "Jack Benny" is proposed, and more details can be found there.

As the possibilities of Arduino IDE, when it comes to working with matrices, are very limited, only an iterative approach was implementable. This means iterating the set of matrix equations over and over in a loop, until the difference in the sought gain value between the iterations can no longer be detected—that is, until the solution converges to a certain value.

The validity of the Jack Benny algorithm and subsequently of this method, has been confirmed in [18], by comparing the results acquired by this algorithm with the native function of MATLAB known under the name `dlqr()`. The `dlqr()` is used for exactly the same purpose—acquiring LQR gain by solving the Riccati equation, but by utilizing specialised solvers. There have not been any significant differences between the results acquired by the Jack Benny algorithm and the MATLAB `dlqr()` function [18].

As a confirmation, the comparison has been repeated, this time directly, between the method `getGainLQ()` and MATLAB `dlqr()` function, and the same conclusion has been reached. Thus, the FloatShield method `getGainLQ()` can be dependably used to acquire the gain (`K`) required for LQR experiments, if the system state-space matrices (`A`, `B`) and penalisation matrices (`Q`, `R`) are known.

More details about the equation solved by this function can be found in Chap. 5.3, concerning LQ control.

The second of the two special template functions, was created to suit the needs of experiments concerning the system control in a closed loop, where all of the system states have to be known (even in a case they are not directly measured). Its code is presented in A.4. The user can access this method by typing

```
getKalmanEstimate(x,u,y,A,B,C,Q_K,R_K);
```

where the arguments `x`, `A`, `B`, `C`, `Q`, `R` are `Matrix` objects of the `BLA` class, with their respective dimensions—`rows` and `columns`—specified at the time of their definition, and arguments `u`, `y` are regular floating point variables.

Similarly, as in the previous method, the matrices `A`, `B` and `C` represent the state-space matrices of the controlled linear or linearised system—while matrices `A` and `B` store the information about the system behaviour and its reactiveness to the input, matrix `C` stores the information about how the individual system states are converted into the actual measurement. In this case, however, the `Q_K` and `R_K` matrices represent noise covariance matrices of the process and the measurement respectively, and they store information about the imperfections of the state-space model (`Q_K`)—how accurately the model describes the system behaviour—and information about the precision of the measurement (`R_K`)—about the deviations and the noise in the sampled signal from the sensor. The argument `y` then represents a value of currently measured output from the sensor, and argument `u` represents system input provided to the system. Finally, argument `x` represents the vector of system states, where system states from the previous sample were stored, and where the function saves the values of currently estimated states.

As the name of the method suggests, it is an implementation of the Kalman filter algorithm used, in this case, for estimating internal system states of the linearised single-input single-output (SISO) system, which for various reasons could not be directly measured.

Knowledge of all of the system states, is crucial in advanced control algorithms that are based on the state-space model of the system—for example, the aforementioned linear quadratic regulator (LQR) or a model predictive control (MPC) algorithm, that will be more closely explained in the following chapters.

The method was inspired by a custom-made MATLAB function proposed in [19], which was created to be implemented as a part of the Simulink model.

To simply describe the algorithm behind `getKalmanEstimate()`, the system state estimation process can be divided into two stages.

First, the initial "a priori" state estimate $\hat{\mathbf{x}}_{(k)}^{-}$ is calculated based on the estimated values of system states from the previous sample $\hat{\mathbf{x}}_{(k-1)}$, on the known current system input $\mathtt{u}_{(k)}$ and on the known system state-space matrices (`A` and `B`).

Second, the final "a posteriori" corrected state estimate $\hat{\mathbf{x}}_{(k)}$ is calculated through the known system state-space matrices (`A` and `C`) and known noise covariance matrices (`Q` and `R`). Based on them, a correction coefficient also known as Kalman gain is calculated, that is used with the known value of current state measurement $\mathtt{y}_{(k)}$, to update the initial "a priori" state estimate $\hat{\mathbf{x}}_{(k)}^{-}$.

In even simpler terms, in the first step, the state-space model is used to calculate expected states $\hat{\mathbf{x}}_{(k)}^{-}$ based on the states from previous step $\hat{\mathbf{x}}_{(k-1)}$, and in the second step, this estimate is corrected, based on actual measurement of the system output $\mathtt{y}_{(k)}$ into the final estimate $\hat{\mathbf{x}}_{(k)}$.

Note, the k represents current time step while the $k-1$ represents the previous one—it is assumed that this discrete process is being sampled with a constant sampling period, and the parameter k then represents the multiples of this period.

The Kalman filter is a wonderful tool, without which a lot of things would simply not be possible. It can be interpreted as a form of an indirect measurement, as it allows us to get sufficiently accurate values of system states that could not be directly measured. A direct measurement might often not be possible due to the lack of needed apparatus or bad access to the location where the value needs to be measured. The second application of the Kalman filter is in the realm of signal filtering, however, this aspect was not important in our case.

More details about the equations utilized by this function can be found in Chap. 4.3, concerning state estimation.

These were the functions created on, and for the Arduino IDE software platform. All of them can be found in the header file `FloatShield.h` (provided in A.1), and source file `FloatShield.cpp` (provided in A.2), of the FloatShield Arduino IDE library.

3.4 FloatShield API for MATLAB

3.4.1 About the MATLAB

The name MATLAB was created by appending together the initial parts of the words matrix and laboratory. Such combination very fittingly describes this well known software

platform developed by MathWorks company.

With its own programming language, focused on making the matrix and array mathematics a more intuitive and simple process, it became the best friend of every scientist, data enthusiast and engineering student. It can be of use to anyone, who often has to perform matrix operations, or has to otherwise work with large quantities of data.

It provides the user with an enormous multitude of tools and functions, that render the whole process easier when it comes to data analysis, signal processing, system identification, designing control systems and many more. Additionally, there is also an option of working with external hardware—such as Arduino boards.

Despite MATLAB, in contrast with the Arduino IDE is neither open-source, nor freely available, it is an important educational tool that has its place in every engineering field. This is confirmed by the fact, that many of the largest and well known technical universities around the world utilize MATLAB—so that their students and teachers can take advantage of this great software.

To be able to work with Arduino hardware within MATLAB, the user has to make sure that the MATLAB support package under the name "MATLAB Support Package for Arduino Hardware" is already installed.

Then, to make sure the support package has been successfully installed, the user can open MATLAB and connect the Arduino device to the computer. If the installation was successful, a message that the device has been recognized will be displayed within the *Command Window*. In case the message was not displayed, the user can try to set the connection up by typing

```
arduinosetup
```

into the *Command Window*, and then, following the setup guide.

If the `arduinosetup` command is not recognized by MATLAB, that is a sign that the support package has not been installed successfully and the user needs to try to install it correctly to be able to work with Arduino boards in MATLAB.

In contrast to the Arduino IDE, where the program, written in C++ is compiled and uploaded onto flash memory, and then is able to run independently of the connection with the computer, MATLAB executes the program on the Arduino board differently.

Specifically, MATLAB does not execute the program on the Arduino board in the true sense at all. What it really does is, that it uploads a server code onto the flash memory of the Arduino board, and then, communicates through the USB connection and through the uploaded server with the Arduino board. It could be even said, that the program executes directly on the development computer, and only then is the information transmitted by the USB connection, and interpreted by the Arduino board. Effectively, the Arduino board becomes a so-called measurement card.

This chain of events has to repeat with every call of the function in every sample, and in case the data are also being read from the Arduino board, the process has to repeat twice in one iteration—but the second time in reverse order. As a result of this, and also because of the fact that the Serial Communication Protocol was not designed for high speed data transmission, a code providing the same functionality written in Arduino IDE will in every case get executed faster than its equivalent written in MATLAB.

It can be stated, that the C++ code on the Arduino board is executed in real-time, while MATLAB instructions provided through the server to the Arduino board are not—they are delayed.

This fact might not play an important role for a regular user, as the communication delay is at most in tens of milliseconds, but if the goal is very quick and responsive control of a mechatronic system, this might become the limiting factor.

Despite the communication between MATLAB and Arduino being heavily limited by the USB Protocol, the support package allows us to run the same experiments that are possible with the Arduino IDE, through the use of very similar functions.

3.4.2 FloatShield General Methods

In MATLAB, the basic file type is the MATLAB Script file (.m suffix). The file of this type, however, can have multiple purposes—based on its content it can act as a simple script containing the commands to be executed, as a file defining a function or even as a file defining the whole class including its methods.

The MATLAB programming language is very different from the C++ used in Arduino IDE, as there is no process of compilation before program execution and no preprocessing (not in the same scope). Therefore the language contains no preprocessor directives to, for example, include a library into the script, there is no need for explicit variable type definition, and also, it provides much more intelligible error messages when there is something wrong with the code. It is overall a significantly more user friendly programming language.

Function definitions, however, are being managed somewhat ineffectively in MATLAB, as each externally defined (meaning that it is not defined directly in the script) function has to be defined in a separate file that has to have the same name as the function that it stores. What this means is, that for example if 10 new functions had to be defined, 10 corresponding new MATLAB Script files to store each one of them would have to be created. On the other hand, in the C++ programming language a single header file may contain as many functions as necessary.

This, thankfully, can be mitigated by defining a class, which is not surprisingly a functionality that MATLAB shares with the C++ language. In such object-oriented languages, the user can define as many functions as necessary, however they will have to belong to a class and therefore have to be used through its object—as methods. The class is defined in the same type of file—MATLAB Script file—and the file has to have the same name as the class it defines.

Note, that the method is just a function that is defined as a part of the class and then used through the class object—there is no real difference between the function and the method, except the way of how or rather where, they were defined.

As there is no preprocessing of the code involved (at least not in the same scope as in C++), the preprocessor directive for including external code is not being used here. The inclusion of external functions and classes in MATLAB is done through the so-called MATLAB Search Path.

The MATLAB Search Path is essentially just a list of paths to the files, that MATLAB looks through when, for example, a function is used in a program. Only, if the path to the folder where the file containing the definition of the used function is stored is part of

the MATLAB Search Path, MATLAB is able to recognize this function by its name and execute its content. If the path to the file containing the function was not part of the MATLAB Search Path MATLAB would throw an error that the function used was not recognized.

This way of looking for functions, classes and other files in MATLAB is probably the main reason why the functions have to be defined each in a separate file; why it is important that the name of the function corresponds to the name of the file it is stored in, and why the names of the files are very important—since MATLAB will look for them by their filename.

However, the user of the AutomationShield library does not have to manually cross-check that all important files are visible and accessible by MATLAB—as there is an installation script present in the AutomationShield library folder which does exactly that.

All that the user is required to do to gain access to all of the AutomationShield functionalities in MATLAB is, to run the script contained within the AutomationShield library folder called `installMatlabAndSimulink` either directly by opening it and pressing the *Run* button, or by typing

```
run installMatlabAndSimulink
```

into the MATLAB *Command Window* while the current folder is the AutomationShield library folder. This command ensures that all of the folders within the AutomationShield library will become visible to MATLAB.

Note, that if the user, for some particular reason, wants to add only the folders concerning MATLAB files, there is a dedicated script for that in the "matlab" folder within the AutomationShield library under the name `installForMATLAB`. It is however more convenient to use the `installMatlabAndSimulink` script as it is directly visible after opening the AutomationShield library folder.

The `installMatlabAndSimulink` along with subfolders concerning MATLAB, includes folders concerning Simulink to the MATLAB Search Path, what will become relevant later.

After successfully running the installation script, the user now has access to all of the AutomationShield MATLAB functions, and most importantly (for the purposes of this work), to the FloatShield MATLAB class. The code of the FloatShield MATLAB API can be found in appendix B. Specifically, the FloatShield MATLAB class `FloatShield.m` is provided in B.1.

The FloatShield device can then be started by using

```
FloatShield = FloatShield;
```

directly followed by

```
FloatShield.begin('portName','boardName');
```

In the first line of code, the object of the class `FloatShield` is created and it is named `FloatShield`. This initialization is required for us to be able to access all the class methods through this object.

Note, that it was not needed to do so when starting the device in the Arduino IDE, because the object of the class was already created at the end of the header file that was included into the sketch, here it is done manually.

In the second line of code the `begin()` method of the `FloatShield` object is called, and it expects to get two input arguments—the name of the serial port that the Arduino board is connected to and through which it will communicate with MATLAB, and the name of the model of Arduino board used.

For example, if the user wanted to run the FloatShield device on the Arduino UNO board connected to the serial port with name COM4, the command would look like `FloatShield.begin('COM4','UNO');`.

The `begin()` method makes sure that the MATLAB library for the TOF sensor has been successfully installed and begins to upload the server onto the specified board through the specified serial port.

This process can take a while, but after the server has been successfully uploaded, the method initialises the TOF sensor, sets it into high speed continuous mode, and initialises important parameters of the device similarly to the Arduino IDE.

After successfully starting the device, the user can calibrate it by calling the

```
FloatShield.calibrate();
```

method to acquire the extremes of the readings measured by the TOF distance sensor.

These values are obtained by utilizing the same approach as in the Arduino IDE—by averaging a certain known number of distance samples taken at the highest position to get a minimal value, and then averaging a certain known number of distance samples taken at the lowest position of the ball to get the maximal value of readings. Afterwards, those values are stored in the `FloatShield` object that will handle these values throughout the rest of script execution.

Calibration results—the (averaged) extreme values of TOF distance sensor measurements (expressed by default in millimetres)—can be accessed by using the methods

```
minimalDistance = FloatShield.returnMinDistance();
```

```
maximalDistance = FloatShield.returnMaxDistance();
```

The raw distance measurement from the TOF sensor expressed in millimetres, can be accessed by

```
rawDistance = FloatShield.sensorReadDistance();
```

but as the usage of raw measurements is rather impractical, this method is seldom called by the user.

To be able to get this measurement, an equivalent of the TOF sensor C++ library had to be implemented in MATLAB, as all of the commands going through the server to the Arduino board have to originate in MATLAB.

That is a more complex situation than it might seem, as the TOF distance sensor depends on the I2C Protocol to communicate with the Arduino board—what means, that

the I2C communication between the sensor and the board has to be continuously managed through the serial communication between MATLAB and the server.

Thankfully, the I2C Protocol is designed for significantly faster communication than the Serial Protocol, and so this fact does not contribute to a bigger communication delay—the main culprit responsible for the delay is still the long communication chain between MATLAB and the Arduino board realised through the slower USB Protocol.

The raw distance transformed by using the measurements gained during calibration into ball altitude expressed in millimetres is provided by

```
ballAltitude = FloatShield.sensorReadAltitude();
```

which is a much more useful variant of the sensor reading method, as the ball altitude is in most cases the value that has to be monitored and possibly controlled.

The last variant of the sensor reading method transforms the altitude by using the measurements acquired in the calibration process into percentual range and can be called through

```
percentualAltitude = FloatShield.sensorRead();
```

where its output can be interpreted as at 0% the ball being on the base and at 100% the ball being at the very top of the tube.

Note, that exactly as in the Arduino IDE, the methods concerning acquisition of measurements from the TOF distance sensor can be called at most every 20 ms, as that is its physical limit. Calling these methods more often would result in repetitive measurements. The communication delay however, caused by the rather slow USB Protocol, would not allow using faster sampling speeds anyway.

To control the power output of the fan the user can call the

```
FloatShield.actuatorWrite(u);
```

method, that expects input `u` in the percentual range—from 0% up to 100%—based on what, it changes the current power level of the electric motor in the fan—by setting the duty cycle of the PWM signal.

The input value `u` to this method can be then simply thought of as duty cycle of the fan itself, or rather directly, the power level of the fan.

To utilise the functionality of the potentiometer runner built into the FloatShield device, the user should use

```
percentualReference = FloatShield.referenceRead();
```

method, outputting the current position of the potentiometer runner expressed as a number in the percentual range of 0%–100%, which allows the user to use the potentiometer

runner as a means of referencing the ball position measured by the method `sensorRead()`.

While in the MATLAB FloatShield API (`FloatShield` class presented in B.1), there are less functions than in its Arduino IDE counterpart, it can be clearly seen that all of the functions that these two APIs do share are identical, either when it comes to their names, or the way they can be called by the user through the `FloatShield` object. The only difference of course being their internal code, that is written using different commands and syntax.

There are currently no release specific functions in the MATLAB FloatShield library such as those in Arduino IDE FloatShield library, as, at the moment, the only release other than the original FloatShield R1—the FloatShield R2—has been deemed unusable in serious experiments due to the already mentioned unwanted behaviour brought by its fan.

As there was no intention of employing it to get any relevant results, trying to implement the RPM reading and writing methods for the FloatShield R2 into MATLAB FloatShield API was not a priority.

3.4.3 FloatShield Special Functions

As it was already stated in Chap. 3.3.4, while working with the FloatShield R1 device it was needed to build specialised functions to serve the particular purposes of more complex experiments.

MATLAB, however, in contrast with the Arduino IDE, is by default better equipped for handling many complicated tasks, for example, concerning lengthy matrix operations. For these the Arduino IDE needed appropriate external library, and it was still not as user friendly. Matrix operations pose no threat to MATLAB, as it was literally designed to be able to efficiently handle them. This allows us to create more complex algorithms in MATLAB in less time, and then utilise only the acquired results in the Arduino IDE.

The sophistication of MATLAB can be illustrated, for example, on the algorithm to calculate the gain for linear quadratic regulator (LQR)—while in the Arduino IDE it was necessary to devise a complex and imprecise iterative algorithm, in MATLAB the user can simply call

```
K = dlqr(A,B,Q,R);
```

what is a MATLAB built-in function created for the very purpose of calculating LQR gain. It also has to solve the Riccati equation, but it does so, through more effective means than by the iterative method.

As in the case of `getGainLQ()`, the matrices `A` and `B` represent the state-space matrices of the controlled linear or linearised system, the matrices `Q` and `R` represent the penalisation matrices of the system states and input respectively. The output matrix `K` represents the gain used in the LQR equations.

The equivalent of the second special method `getKalmanEstimate()` from Arduino IDE in MATLAB, was actually originally created in MATLAB and only then implemented for Arduino IDE. Its code can be found in appendix B.2.

This function can be in MATLAB accessed by calling

```
[xEst,yEst] = estimateKalmanState(u,y,A,B,C,Q_K,R_K,xIC);
```

where the arguments `A`, `B`, `C` are state-space matrices representing the FloatShield system, `Q_K`, `R_K` are noise covariance matrices representing the quality of the model and measurements, and `u`, `y` are the current system input (power to the fan) and output (sensor distance measurement) respectively. The last argument `xIC` is optional, and can be used to set the initial values of estimated states in case, that the estimation function should not assume zero initial states.

The function then returns two arguments—a vector of estimated states `xEst`, and the estimated (may also be interpreted as filtered) output `yEst`—that can also be found within the estimated state vector, and is provided by the function only for practical reasons.

As it has been already mentioned in Chap. 3.3.4, this function implementing a Kalman filter was inspired by a custom made MATLAB function proposed in [19], that was created to be used as a part of a Simulink model. It was then modified to fit the needs of the FloatShield device and also any other AutomationShield device.

It utilizes exactly the same principle as the `getKalmanEstimate()`—the estimation is a two step process where, first the initial estimate is calculated based on the state-space model, and then the initial estimate is corrected into its final form based on the provided noise covariance matrices and the measured system output.

More details about the equations utilized by this function can be found in Chap. 4.3, concerning state estimation.

These two aforementioned MATLAB functions were the equivalents of the respective methods in Arduino IDE, but the ease with which MATLAB is able to handle complex matrix operations, data processing and also data visualisation, allowed to create additional useful tools that the FloatShield device needed.

For example, a plotting function that manages the formatting of plotted data that were acquired in control experiments. The code it utilizes is provided in appendix B.3. It can be called by typing

```
plotResults(dataToPlot)
```

where the input argument `dataToPlot` can be either a variable directly from the MATLAB *Workspace* containing the data, a text file containing the data, or a specialised MATLAB file to store variables and data (.mat suffix). The only condition is, that the data shall be stored in a matrix format, or a list with three columns, where each line is representing `[reference, output, input]` values, in this specific order.

This format requirement has to be met by `dataToPlot` to allow its plotting by the `plotResults()`, what should pose no problem as the function was specifically created

to provide the user with a uniform way of plotting results from control experiments—where all of the three mentioned values are commonly present.

Another very convenient function makes the inclusion of important matrices from MATLAB to the Arduino IDE an easy task. Its source code is provided in appendix B.4. By typing

```
printBLAMatrix(mat1,'name1',mat2,'name2',...,'fileName')
```

the user can save any number of matrices from the *Workspace* under specified names, in a format compatible with the BLA Arduino IDE library. The matrices are stored into a header file named `fileName`, with an option to not specify the `fileName`, in which case, the matrices will be saved to header file `BLA_Matrices.h` by default.

A single requirement has to be met to successfully save the matrices—it is necessary to abide the correct order of input arguments. First comes the matrix variable from MATLAB *Workspace*, and after it comes the name under what the matrix will be imported into Arduino IDE as a BLA matrix.

Matrices saved from the MATLAB *Workspace* into a header file this way, can then, utilizing a preprocessor directive, be directly imported into the Arduino IDE, where they can be used as a BLA matrix variables.

This allows the user to comfortably include key matrices acquired by complex means in MATLAB, in Arduino IDE sketches.

The last two functions worth mentioning were greatly needed in order to perform experiments concerning MPC.

First of them can be called with

```
[H,G,F] = calculateCostFunctionMPC(A,B,np,Q,R,P);
```

Where `A`, `B` are linear state-space matrices—representing the behaviour of the system—`np` is the prediction horizon—representing how many steps into the future the controller should look when calculating system input—and `Q`, `R`, `P` are penalisation matrices of states, input and the final state respectively—that can be used for influencing the behaviour of the controller.

The source code of this MATLAB function can be found in appendix B.5. Its purpose is to calculate the matrices `H`, `G`, `F` that define the so-called cost function of the system.

A detailed definition of the cost function together with the minimisation problem it entails, will be more closely described in Chap. 5.4, concerning MPC.

Second of the functions, closely relates to the first, and is also used within experiments regarding MPC. It is used by typing

```
[Ac,b0,B0] = applyConstraintsMPC(uL,uU,xL,xU,np,A,B);
```

where as in previous case `A`, `B` are linear state-space matrices and `np` is the prediction horizon. Additionally the parameters `uL`, `uU` represent the lower and upper limit of system inputs, while the parameters `xL`, `xU` represent the lower and upper limits of internal states.

This function is used to formulate input and state constraints within the cost function—it sets limits on values of certain parameters that the solver must respect while looking for the minimum of the cost function. These constraints are applied through matrices `Ac`, `b0` and `B0`, which define a constrain inequation. The source code of this function can be found in appendix B.6.

More details about the constraints and their inequation will be provided in Chap. 5.4, concerning MPC.

These two aforementioned functions—the `calculateCostFunctionMPC()` and the `applyConstraintsMPC()`—were heavily inspired by the book [20], where the authors provide a more detailed insight into model predictive control, including many examples and custom functions executable directly in MATLAB.

While most of these MATLAB special functions are not directly related to the FloatShield device itself, they are presented here, as they were created to serve the FloatShield, and each one of them was, and still is, being used to either allow special control experiments to be run on the device, to make the resulting graphs look neater, or to allow easier matrix transfer between MATLAB and Arduino IDE.

Despite being created primarily for the purposes of FloatShield, they are now a part of the AutomationShield MATLAB API, where they can be utilised by any and all of the other AutomationShield devices, thus making the entire library more richer.

3.5 FloatShield API for Simulink

3.5.1 About the Simulink

Even though declaring Simulink to be a separate software platform along with Arduino IDE and MATLAB at the beginning, it is not exactly so. To be more precise, Simulink is a part of the MATLAB software environment, and is rather a module or extension, than a separate software platform.

Its interface does not look at first glance very different from the one used by the MATLAB, however, it utilizes significantly more graphical elements. Such a GUI results practically in no code typing, but rather "building" the code from algorithmic blocks. Individual functions, variables and tools are stored inside these blocks, often created with comprehensive GUI. They are then connected with lines, representing the "flow" of information around the so-called model.

The Model is a term referring to a Simulink file containing the program built of blocks. Its full name is a Simulink Model, and it uses the .slx suffix for its files.

As its name partly suggests, Simulink specializes in modeling and performing simulations of systems, while also analyzing the ongoing processes. Its key benefit is to try out ideas and changes harmlessly before actually realising them on hardware, and as it builds upon MATLAB, it offers most of the tools already present in MATLAB, and even more.

Simulink does not natively support Arduino hardware, however, as it is in case of MATLAB, there is a specialized package for that purpose. To be able to work with Arduino hardware within Simulink, the user must ensure, that the Simulink support

package under the name "Simulink Support Package for Arduino Hardware" is already installed.

To make sure the support package has been successfully installed, the user has to open MATLAB and connect the Arduino device to the computer. If the installation was successful, a message will be displayed in the *Command Window* stating, that the device has been recognized and can be used by Simulink.

Note, that it is highly recommended to have the "MATLAB Support Package for Arduino Hardware" installed at the same time.

Contrary to MATLAB, where the program gets executed on the computer and only then are the hardware instructions communicated through serial connection with server uploaded on Arduino board, Simulink offers an option of running the program very similarly to the Arduino IDE—directly on the Arduino board through a compiled C/C++ code.

An embedded program for the Arduino board is created as a Simulink model by using built-in Simulink blocks and the support package, where the interaction with the board is managed through specialised blocks obtained from the support package. When the program is ready, it gets (similarly to the Arduino IDE) compiled and then uploaded onto the Arduino board, where it runs as a standalone program regardless of the connection with the computer. This ensures, that there are no additional delays in the execution, as it was in the case of MATLAB. However, Simulink brings completely new kinds of problems, specifically, concerning code compilation. The Simulink compiler is very unintuitive and inefficient, what often causes complications when building a more complex program. It often happens, that after successful compilation, the program takes up too much memory space of the Arduino device. The reason for this, is that the Simulink has to first transcribe the blocks into the C++ language, and only then compile the resulting code—while the transcribed code is not optimised for the use in embedded devices. Then, uploading even a few simple blocks onto the board may take up more than half of the memory available for the program.

When creating a program intended to be run on Arduino boards in Simulink, it is important to specify the particular model of the Arduino board that is being used, in the Simulink model settings. Based on that, Simulink will know how exactly the code should be compiled for the board. Also, one must not forget to use the external execution mode instead of normal, as the program has to be executed externally, on the physical board itself (this option in execute mode selection is available only up to Simulink version 2019a). The last essential thing to be keep in mind is, to specify the fixed step size in the solver settings. Otherwise, the solver might automatically adjust the step size, and the results thus obtained may have a questionable value.

3.5.2 FloatShield General Blocks

The libraries containing functions are not present in the same form in Simulink, as they are in Arduino IDE. There is no "code" in a sense of lines of commands. However, bearing in mind that Simulink uses blocks as means of interaction, there must be libraries containing blocks enabling this functionality.

And it is exactly so—the library in Simulink is just a Simulink Model file (.slx suffix), that is created with the intention of being a library file. Such file, may then contain

custom made blocks, that can offer various useful functionality.

By including the folder in which this Simulink library file is located into the MATLAB Search Path, the blocks become accessible from within Simulink. Then, they can be found using the *Library Browser* in Simulink. If the library is not visible after including its location into the Search Path, it might be needed to refresh the *Library Browser* by pressing the F5 button on the keyboard.

The AutomationShield Simulink API has to be installed exactly this way—by including it into the MATLAB Search Path. However, the library user does not have to do this manually, as the library provides a convenient way of installation by running the `installMatlabAndSimulink` file within the AutomationShield library.

Note, that if the user for some particular reason wants to add only the folders concerning Simulink files, there is a dedicated script for that in the "simulink" folder within the AutomationShield library under the name `installForSimulink`.

If the `installMatlabAndSimulink` has been already run when installing the AutomationShield library for MATLAB, it is no longer necessary to run it again for Simulink.

After successfully running the installation script the user now has access to all of the AutomationShield Simulink blocks, which should become available in the *Library Browser* within the Simulink model.

To access the measured data from the TOF distance sensor in Simulink, the Simulink block `Sensor Read` is used, which is illustrated in Fig. 3.1.

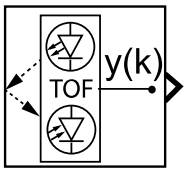


Figure 3.1: Sensor Read Simulink block.

The user can obtain the position of the object from distance sensor in a specified form through the `Sensor Read` Simulink block. This means either in raw distance in millimetres, altitude in millimetres or in altitude in percentages, while the choice can be made within the settings of the `Sensor Read` block.

Within these block settings (accessible by simply double clicking on the block), the user can specify the sampling period of the block, and may also manually specify the expected extreme values of the measurement, which are normally acquired during the calibration process.

To be able to access the TOF sensor registers, the `Sensor Read` block utilises a Simulink functionality called an "S-Function". An S-Function, or rather S-Function block combined with the S-Function Builder block in Simulink, enables the user to create blocks that implement external C and C++ code.

As the TOF distance sensor is heavily dependent on the manufacturer provided C++ library, to make the distance sensor work within Simulink, the utilisation of the S-Function block was practically inevitable. The S-Function Builder block serves as a convenient tool for creating S-Function blocks, as thanks to its user friendly graphical user interface (GUI)

the whole integration process is more easily understood and managed.

Controlling the power level of the FloatShield device can be done through the `Actuator Write` Simulink block, which is depicted in Fig. 3.2.

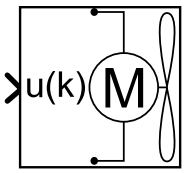


Figure 3.2: Actuator Write Simulink block.

By using the `Actuator Write` Simulink block the user is able to regulate the fan power output the same way it was possible in Arduino IDE and MATLAB—the block expects to get a single value in the percentual range 0%–100%, which represents the current power level of the fan.

To provide this functionality, the block makes use of the PWM block present in the Simulink support package for Arduino boards, through which, it is possible to regulate the duty cycle of the PWM signal on the given PWM-capable pin.

Accessing the reference provided by the built-in potentiometer is done by the `Reference Read` Simulink block, which is shown in Fig. 3.3.

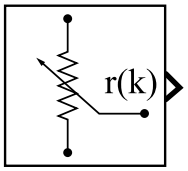


Figure 3.3: Reference Read Simulink block.

Aside from defining the sampling period of the `Reference Read` Simulink block within its settings, the user can also choose the format of its output signal—whether it should return the position of the potentiometer in percentual range 0%–100%, in voltage range 0 V–5 V or as a direct reading from the ADC—as a value in the range of 0–1023 (in case of a 10-bit ADC resolution).

To represent the FloatShield device inside a closed loop scheme more illustratively, the `FloatShield` Simulink block depicted in Fig. 3.4 can be used.

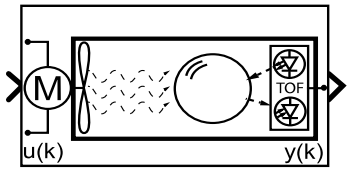


Figure 3.4: FloatShield Simulink block.

The `FloatShield` Simulink block is a more accurate representation of the FloatShield device than could be achieved by simply using the individual `Sensor Read` and `Actuator Write` Simulink blocks, as in addition to combining the functionalities of these two blocks within a single one, the `FloatShield` Simulink block additionally implements the process of the device calibration.

Along with defining the sampling period of the `FloatShield` Simulink block, the user can specify the number of calibration samples taken. This means, from how many samples should the minimal and maximal measurements, required for more precise output mapping, be averaged. Additionally, the format of the block output can be selected—the ball position inside of the tube can be expressed either in altitude in millimetres, or as percentual altitude.

As in the case of the `Actuator Write` block, the `FloatShield` block expects input in the form of percentages—in the range of 0%–100%. This represents the amount of power output the fan should produce.

The whole calibration process along with sensor reading and actuator writing functionalities, is being coordinated thanks to the Simulink block "MATLAB Function". The functionality of this block is realised through MATLAB code. Utilizing a series of written MATLAB commands can be, in many cases, more practical and straightforward, than if the same functionality should be built only by using a series of interconnected Simulink blocks.

In case there is limited memory, this can be mitigated by the use of MATLAB Function blocks, which, in comparison with Simulink native blocks take less memory space. This was confirmed by comparing two simple blocks with identical functionalities, while one being native Simulink block and the created by the MATLAB Function block. The block created by MATLAB code took up less memory after being compiled and uploaded onto the Arduino board.

While such a program size enhancement might not seem significant, in the case of limited memory on some boards (for example the Arduino UNO), even such small changes might mean the difference between being able to run the program on the device just fine, or not being able to run the program at all. Because of that, the MATLAB Function block is used as a part of the `Sensor Read`, `Actuator Write` and `FloatShield` blocks, to allow more precise control of the individual algorithmic modules as well as to save up memory.

3.5.3 FloatShield Special Blocks

As with the previously mentioned software environments, it was needed to create specialised blocks enabling certain actions and experiments to be done in Simulink as well.

For example, in reference tracking experiments, to specify the reference the user can pick a Simulink block called `Reference` which is shown in Fig. 3.5. It provides a very convenient way of supplying the reference during the simulation, where the reference can be periodically changed based on a defined pattern.

After specifying the sampling period of the `Reference` block the user can explicitly enter a vector of references that should be provided during the simulation.

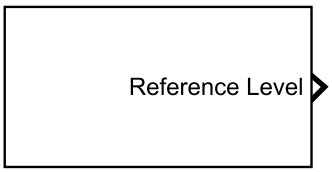


Figure 3.5: Reference Simulink block.

In addition to that, the value of section length can be set, meaning, for how many samples should each element of the reference vector be supplied by the block. The block then ensures that the simulation will last precisely the amount of samples equal to the section length multiplied by the length of the reference vector, and after reaching this time (expressed in samples) the simulation will stop.

If the user wants to stop the simulation manually, there is another parameter in the `Reference` block settings, specifying what should be done after reaching the end of the reference vector. The user can either choose to start the referencing sequence from the beginning (the block will output reference values indefinitely until the simulation is stopped manually), or to repeat the reference sequence only a specified number of times (the referencing train will start from its beginning for a specified number of times and after reaching the final sample it will stop the simulation).

The last input parameter of the `Reference` block is the optional offset time reserved for auxiliary processes—the user can specify the time in seconds during which the `Reference` block will be inactive at the beginning of the simulation, as it expects that processes, such as calibration, are being executed. Without specifying this parameter, the block would work as well, but the first reference value would be correctly provided for a smaller number of samples than specified by the section length.

There are no units specified with the reference values—they are simply numerical values—what makes the block universal and useful in all experiments requiring a changing reference sequence.

Another example of a purpose-specific block is the case of the linear quadratic regulator (LQR) in Simulink, as there is no native Simulink block for this purpose. The probable reason for its nonexistence is the relative simplicity of LQ control implementation. However, if it would include an integrator state, its application becomes more complex.

The implementation of LQR along with an additional integrator state is realised within the `FloatShield LQ Regulator (+Integrator)` Simulink block, that is depicted in Fig. 3.6.

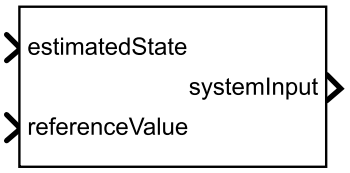


Figure 3.6: LQ Regulator Simulink block.

This block, asides from the sampling period, has only one additional input parameter, and that is the LQR gain matrix.

Its purpose is to calculate the required system input using a specified gain matrix, while taking into consideration the current error (the difference between the actual value of the controlled parameter and its reference), and also error from past (stored by the integrator state).

It has been created specifically for the FloatShield device, as in the case of FloatShield, not all of its states are being measured, and therefore, they need to be estimated. The block then requires to get all three system states as inputs, while calculating the fourth one automatically.

This process will be more thoroughly described in Chap. 5.3, concerning LQR.

Because of the problems with insufficient memory, when trying to build certain experiments in Simulink, more memory friendly (`light`) versions of the blocks `FloatShield` and `Reference` have been created.

The only two input parameters of `FloatShield(mm)(light)` are sampling period and calibration time in seconds, while it can output object position only in the form of altitude in millimetres.

When it comes to the three input parameters of `Reference(light)`, they are values of the reference vector, section duration in seconds and sampling period.

Both of these (`light`) blocks are built to provide sufficient functionality while using only a minimal number of the native Simulink blocks and implementing no MATLAB Function blocks. It was found, that the persistent variables that are used in MATLAB Function blocks to keep track of certain parameters during simulation, have a negative impact on the memory, if used excessively.

Therefore, a balance had to be found between using MATLAB Function blocks, and Simulink native blocks. MATLAB Function blocks were used only in necessary cases, where it would be too complicated to use Simulink blocks.

Unfortunately, this balance has to be made in every Simulink model that will be uploaded onto an Arduino board individually, as the Simulink compiler is unpredictable at times and it often requires a lot of trial and error, until the process of compilation is successful.

Those were the functions/methods/blocks of the FloatShield API within the AutomationShield library.

The difference between functions/methods/blocks labeled as general and those labeled as special was sometimes obvious and sometimes subtle, but in most cases, the general functions/methods/blocks all serve only the FloatShield device and are related to the device basic functionality, while the special functions/methods/blocks are often more universal and serve a more concrete purpose, and are usable by other AutomationShield devices. While the general functions/methods/blocks are in many cases essential for the operation of the FloatShield device, the special are often not. These are usually not related to the FloatShield device itself but rather to some functionality, through which they enable its use in special experiments.

The code of the aforementioned software tools is presented in appendices A and B.

Chapter 4

Modeling, Identification and Estimation

The FloatShield device, which embodies an air levitation experiment, exhibits a highly nonlinear behaviour. Its nonlinearity can be observed already, when testing the basic device functions—it can be experienced by the user directly, even before performing elaborate measurements and tests.

For example, certain patterns in the behaviour of the ball can be recognized by running a simple open loop example, present in the `FloatShield_OpenLoop` Arduino IDE sketch file (.ino suffix). This example combines three basic functions, used to: get reference value from potentiometer runner; set power input to the fan; get measurements from the sensor. As a result, the user is able to control the power input to the fan using the potentiometer runner, while the effects of input change on the ball behaviour can be observed either on the monitor, or directly on the device. After uploading the example, the user can check the validity of the statements presented below. Its C++ code can be found in appendix A.5.

In the case of FloatShield, the same change of the power input does not always result in the same change of ball position. Specifically, if the fan power input is anywhere from 0%, up to approximately 35%, the fan spins and creates a stream of air, but of insufficient intensity. In this power range, the stream is not strong enough to lift the ball off, which then just hovers, at most, a few millimetres above the base, without going any higher.

However, when the power input is changed by just additional 5%—to approximately 40% of fan power input—the ball starts to slowly rise and, in a matter of seconds, ascends to the very top of the tube, and stays there.

It can be seen, that while the power input from range of 0% up to approximately 35% has no or very minimal effect on the ball, the power input from range of approximately 40% up to 100% guarantees, that the ball will rise to the very top of the tube. The only difference being the rising speed—with 40% the ball rises rather slowly, while at 100% the ascend is almost instant.

Then, making the input changes within the 5% input power range (from approximately 35% up to approximately 40%), seems like the only way of balancing the ball in a stable position. This is actually not very far from the truth, as it will be later obvious.

Note, that the word "approximately" is used here, as no two same devices are really the same—the values of the boundary input powers might slightly vary. However, the inherent nonlinear behaviour of the devices is still the same.

Because of this fact, the FloatShield system can be safely labeled as a nonlinear system, as the aforementioned relationship between the power input and the ball position does not satisfy the superposition principle. This principle states, that for linear systems the response caused by two or more stimuli, should be equal to the sum of the responses that would have been caused by each stimulus individually [21]. However, the ball altitude at the power level 40%, is not the same as the sum of ball altitudes at power levels of 5%, and 35%—therefore, the FloatShield device is not a linear system.

The major cause of the nonlinearity in the FloatShield device is, the medium it uses as a means of moving the ball—air. However, it is not a "fault" of the air itself, as it would be the same with any other fluid. It is simply in the nature of fluids, which often do not flow in straight lines (laminarly), but tend to form swirls and then flow more chaotically (turbulently). At the same time, the erratic reaction of the ball can be attributed to the delicate balance between the gravitational force, and the air drag force, that are acting in opposite directions.

While the effect of gravitational force is obvious, the value of the drag force depends on many factors. Some of them are constant—such as the exposed area of the ball to the air stream, or density of the air—while others are gradually changing—such as the drag coefficient that is dependent on the Reynolds number, or relative velocity of the ball.

Turbulence emerging within the air flow, can cause abrupt changes in the drag force applied on the ball, while also causing unwanted horizontal movements, visible often in the form of oscillation—thus making the ball behaviour more erratic and unpredictable.

It is an unfortunate fact, that while the power output of the FloatShield device fan can be readily and predictably controlled, and at the same time the current position of the cork ball can be dependably measured, there is no form of influence at our disposal over the native behaviour of the ball floating in the stream of air.

If the behaviour itself can not be changed, the most logical course of action is to try to understand it as deeply as necessary, so that it will become possible to work with it or even take advantage of it.

4.1 Modeling

The process of modeling, or more specifically the process of designing and looking for a mathematical model of the system, is an important step towards being able to work with, and also better understanding the physical system. Such description of the real physical system in form of mathematical expressions, can serve as a system representation within computer simulations.

Mathematical models often describe the relationship between the system inputs—for example voltage or torque—and system outputs—for example temperature or rotations per minute. System input parameters usually represent the required effort of the actuators, while the system output parameters represent the actual values of observed internal parameters of the system, acquired by the sensors.

In simple terms, this relationship expresses, how the system reacts (output), to the provided external stimuli (input). If the mathematical model is successfully set up, it can be then used to simulate and even predict the system behaviour—what has an enormous use in the feedback control.

While the idea of mathematical model can sound like a solution to all problems, in real life, setting up a proper and accurate mathematical model of a real system is rarely a simple task. The main problem of any mathematical model is, that its description of the real physical system will always be only an approximation. While the model can sometimes be very accurate in a broad range of system parameters, other times it might describe a system sufficiently, only under specific conditions and in specific intervals of parameters.

However, this is not the issue of models, but more so of mathematics itself, as it serves us to try to describe the real world phenomena in terms of written equations and expressions. While there may be an equation that perfectly describes some part of our reality, in that case, it will be a very theoretical one, without actual practical use. On the other hand, the equations actually used by engineers are in most cases very simplified versions of those theoretical equations. In these, most of the original parameters are either left out completely or are represented by table constants—what is not necessarily a bad thing, as a good enough model is always better than no model.

Generally, a very intuitive two step approach can be used when looking for a practically usable mathematical model of a system. The first step is, to look for the equations based on first principles (postulates), that could describe the whole system or, at least, a part of it. One has to identify the processes occurring within the system, and if it is not possible to describe them as a whole, the goal should be trying to separate them and describe them one by one. Using first principle equations means, that only a generally accepted (regarded as true) theoretical equations and laws of science should be used when describing parts of a system—for example Newton's laws for mechanical systems or laws of thermodynamics for thermal systems. As it was mentioned before, while these theoretical laws and equations are very accurate, they are in most cases expressed in a practically unusable form. Therefore, after successfully describing the relationship between inputs (to) and outputs (of) the system, they have to be simplified.

Then, the goal of the second step is a correct realisation of this simplification. In this process, some of the accuracy will be lost, but the amount lost will depend on the correctness of the simplification. It is crucial to find the right balance between making the mathematical description practically usable, and keeping most of its original accuracy intact. This is done through various methods, such as linearization of nonlinear parts of functions by Taylor series expansion around specific working points while omitting higher derivations, and lumping the factors that have only limited impact on the system into constants. Additionally, the continuous expressions are discretized, so they can be implemented into computation hardware.

It was expected—as the main process in the FloatShield device utilizes air flow—that the equations used to describe the system will have to be looked for in the branch of aerodynamics.

However, as it was already mentioned in Chap. 1.3, the FloatShield device is not the first of its kind—there have been many attempts on creating very similar devices all over the world. Therefore, other similar works have been studied and reviewed when looking for the best model for the FloatShield, to get the idea of what would be the most suitable approach—most notably in [5, 7, 9, 10, 15, 17].

It turned out, that our starting assumption of needing an descriptive equation concerning aerodynamics was correct, as in most papers the air drag equation was present in various shapes or forms. Here, the model introduced in [7] will be described more closely, as it was deemed the most compatible with options and the needs of our system.

The basic form of the air drag equation for the air levitation system, is shown by the following equation

$$F_{D(t)} = \frac{1}{2} C_D \rho A_b \left(v_{(t)} - \dot{h}_{(t)} \right)^2, \tag{4.1}$$

where:

- $F_{D(t)}$ is the drag force applied on the ball by the stream of air at time t;
- C_D is the so-called drag coefficient;
- ρ is the density of air;
- A_b is the area of the ball exposed to the upwards air flow;
- $v_{(t)}$ is the velocity of the air inside the tube at time t;
- $h_{(t)}$ is the position (altitude) of the ball inside the tube at time t.

While the parameters ρ and A_b can be treated as constants, the velocities of the air $v_{(t)}$ and the ball $\dot{h}_{(t)}$ inside the tube (and C_D dependent on them), will be constantly changing. Based on this equation, it can be assumed, that these velocities will play an important role in a correct model of the system.

Note, that C_D is in reality dependent on the Reynolds number—and so it changes with the change in air flow velocity, and ball velocity. However, C_D is (for our purposes) considered to be a constant, as an acceptable simplification.

Besides the drag force caused by the air flow—acting in the upward direction—only one other force acts on the ball—the gravitational force. The gravitational force acts in a opposite direction as the drag force—downward direction—and its magnitude depends only on the mass of the ball and the gravitational acceleration—which can be (for our purposes) treated as a constant value of $9.81\,\mathrm{m\,s^{-2}}$.

The gravitational force can be expressed by a rather simple equation

$$F_G = mg, \tag{4.2}$$

where:

- F_G is the gravitational force applied on the ball;
- m is the mass of the ball;
- g is the gravitational acceleration.

With all (both), of the forces acting on the ball known, and expressed by Eq. (4.1) and Eq. (4.2), Newton's second law of motion can be applied. This law states, that the vector sum of the forces acting on an object, is equal to the mass of that object, multiplied by the acceleration of that object.

The result of applying this law on the case of FloatShield system (while assuming the upward direction on the vertical axis as positive), is equation

$$F_{D(t)} - F_G = m\ddot{h}_{(t)} = \frac{1}{2} C_D \rho A_b \left(v_{(t)} - \dot{h}_{(t)} \right)^2 - mg. \tag{4.3}$$

Furthermore, the parameters that are assumed to be constant, can be lumped into a single unknown—let us call it α, and let it be equal to $\alpha = \frac{1}{2} C_D \rho A_b$.

By applying this substitution, and by dividing both sides of (4.3) by the mass of the ball, the following equation will be obtained

$$\ddot{h}_{(t)} = \frac{\alpha}{m} \left(v_{(t)} - \dot{h}_{(t)} \right)^2 - g. \tag{4.4}$$

Equation (4.4) (by introducing only small approximations) describes the vertical movement of the ball in relation to the air velocity quite accurately.

It is important to realise, that the levitating ball will be in steady state when it does not move—in that situation the $\ddot{h} = \dot{h} = 0$ expression should be true. In this state, the ball is stable and staying at the same vertical position, what means that the air velocity retains its value—there is a specific air velocity for ball at equilibrium in a specific position.

Let us call this air velocity v_{eq}, and let it be known as air velocity at the equilibrium. Then, the Eq. (4.4) will change into

$$g = \frac{\alpha}{m} \left(v_{eq} \right)^2, \tag{4.5}$$

in steady state. The value of the theoretical equilibrium air velocity can be then extracted from (4.5), by

$$v_{eq} = \sqrt{\frac{mg}{\alpha}}. \tag{4.6}$$

After that, by substituting the gravitational acceleration in Eq. (4.4) by Eq. (4.5), the dynamic equation can be finally given as follows

$$\ddot{h}_{(t)} = g \left(\left(\frac{v_{(t)} - \dot{h}_{(t)}}{v_{eq}} \right)^2 - 1 \right). \tag{4.7}$$

This quadratic differential equation (4.7), represents the relationship between values of individual accelerations and velocities, and thus describes the ongoing movements in the FloatShield system.

It can be safely stated, that this equation describes the main process of the FloatShield device with great accuracy (especially in case that the true value of v_{eq} is known)—the movement of the ball in relation to the air speed produced by the fan.

As it would be very impractical to work with the quadratic form of this equation, our aim is to linearize it—in this case by utilizing a Taylor series expansion.

First, let us substitute the quadratic member with $x = \frac{v_{(t)} - \dot{h}_{(t)}}{v_{eq}}$. Then, the Eq. (4.7), can be written as $f = g\,(x^2 - 1)$. Afterwards, it can be simply linearized around the equilibrium point—where $v_{eq} = v - \dot{h}$ is true and therefore also $x = 1$ is true—using Taylor's approximation of $f_{(x)} \approx f_{(1)} + f'_{(1)}(x-1)$. The linearized version of Eq. (4.7)—and therefore also the linearized version of the FloatShield motion equation is

$$\ddot{h}_{(t)} = 2g\,(x-1) = \frac{2g}{v_{eq}}\left(v_{(t)} - \dot{h}_{(t)} - v_{eq}\right). \tag{4.8}$$

In such a way, a practically usable mathematical model was obtained, expressed in a form of a linear differential equation of motion (4.8). The model should—assuming that the working range is near to the steady state—provide us with the relationship between acceleration and velocities within the system, of sufficient accuracy.

From this model, it is then possible to derive its other forms—for example a form of process transfer function. To acquire this form, an operation of Laplace transformation has to be applied on the differential equation (4.8). Assuming that instead of parameters h and v, only their perturbations from equilibrium δh and δv will be calculated by the model, it is possible to extract the transfer function between the change in air velocity and change in ball altitude, within the s-domain.

This transfer function is defined in the s-domain by

$$\frac{\delta H_{(s)}}{\delta V_{(s)}} = \frac{1}{s\,(\tau_1 s + 1)}, \tag{4.9}$$

where:

- $\delta H_{(s)}$ is the perturbation of ball position from equilibrium expressed in the s-domain;
- $\delta V_{(s)}$ is the perturbation of air velocity from equilibrium expressed in the s-domain;
- s is the Laplace operator;
- τ_1 is the system time constant, while also $\tau_1 = \frac{v_{eq}}{2g}$.

By assuming, that the relationship between the air speed and the fan input can be expressed in the s-domain with a transfer function of similar form

$$\frac{\delta V_{(s)}}{\delta U_{(s)}} = \frac{K_f}{s\,(\tau_2 s + 1)}, \tag{4.10}$$

where:

- $\delta U_{(s)}$ is the perturbation of system input from equilibrium expressed in the s-domain;

- K_f is the gain constant of the fan;
- τ_2 is the fan time constant,

it is possible (by multiplying (4.9) and (4.10)) to acquire the process transfer function of FloatShield, describing the relationship between fan input and the ball altitude. The linearized model of the FloatShield system, in form of a process transfer function relating to the perturbations of output and input is then

$$\frac{\delta H_{(s)}}{\delta U_{(s)}} = \frac{K_f}{s\left(\tau_2 s + 1\right)\left(\tau_1 s + 1\right)}. \tag{4.11}$$

Additionally, by continuing with the perturbation form of the model (gravitational acceleration is omitted in this form), and by utilizing the system time constant τ_1 defined in (4.9), the original linearized model (4.8) can be written as

$$\delta\ddot{h}_{(t)} = \frac{1}{\tau_1}\left(\delta v_{(t)} - \delta\dot{h}_{(t)}\right). \tag{4.12}$$

Finally, by returning from the s-domain back into time domain, the equation (4.10) takes form

$$\delta\dot{v}_{(t)} = \frac{1}{\tau_2}\left(K_f \delta u_{(t)} - \delta v_{(t)}\right). \tag{4.13}$$

Now, based on the equations (4.12) and (4.13) it is possible to formulate the linearized model in a state-space form.

By assuming that the state vector is defined as $\mathbf{x}_{(t)} = \begin{bmatrix} x_{1(t)} & x_{2(t)} & x_{3(t)} \end{bmatrix}^{\mathrm{T}}$, or more specifically $\mathbf{x}_{(t)} = \begin{bmatrix} \delta h_{(t)} & \delta\dot{h}_{(t)} & \delta v_{(t)} \end{bmatrix}^{\mathrm{T}}$, the (4.12) can be rewritten to

$$\dot{x}_{2(t)} = -\frac{1}{\tau_1} x_{2(t)} + \frac{1}{\tau_1} x_{3(t)}, \tag{4.14}$$

and (4.13) can be transformed into

$$\dot{x}_{3(t)} = -\frac{1}{\tau_2} x_{3(t)} + \frac{K_f}{\tau_2} u_{(t)}. \tag{4.15}$$

After introducing an auxiliary state equation $\dot{x}_{1(t)} = x_{2(t)}$, these three state equations can be written together in a matrix form

$$\delta\dot{\mathbf{x}}_{(t)} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & -\frac{1}{\tau_1} & \frac{1}{\tau_1} \\ 0 & 0 & -\frac{1}{\tau_2} \end{bmatrix} \delta\mathbf{x}_{(t)} + \begin{bmatrix} 0 \\ 0 \\ \frac{K_f}{\tau_2} \end{bmatrix} \delta u_{(t)}. \tag{4.16}$$

This matrix equation (4.16), then represents the linearized state-space model of the FloatShield device.

A similar, albeit not linearized and absolute (not perturbational) version of the state-space model—one that retains the quadratic member and also gravitational acceleration—can be obtained by using (4.4) as a second state equation instead of (4.14). This form was also proposed in [15], and consists of the following state equations

$$\dot{x}_{1(t)} = x_{2(t)}, \tag{4.17a}$$

$$\dot{x}_{2(t)} = \frac{\alpha}{m}\left(x_{3(t)} - x_{2(t)}\right)\left|x_{3(t)} - x_{2(t)}\right| - g, \tag{4.17b}$$

$$\dot{x}_{3(t)} = \frac{K_f u_{(t)} - x_{3(t)}}{\tau_2}. \tag{4.17c}$$

Note, that the difference between the two state-space models (4.16) and (4.17) (aside from being in perturbational or absolute form), is only in their second state equations (4.14) and (4.17b). However, the nonlinear model (4.17) requires the utilization of mathematical solvers to be applicable—such as the ones present in MATLAB.

Thus, three practically usable forms of the FloatShield system mathematical model have been acquired, based on the motion equation (4.4) obtained from Newton's second law of motion. Specifically, the models have a form of:

- linearized transfer function (4.11),
- linearized state-space model (4.16),
- nonlinear (quadratic) state-space model (4.17).

4.2 Identification

Although being presented in different chapters, the processes of modeling and identification are not always to be so easily separated.

There are generally three approaches of system identification:

- **White box** system identification assumes, that the identified system can be completely defined based on sufficient knowledge and understanding of its processes. Therefore, it is possible to describe it with the necessary number of complete equations, where all the parameters are known.

  Then, there is no need for additional measurements for acquiring unknown constants and parameters of the model, as everything is known and can be known.

- **Gray box** system identification assumes, that the identified system can be defined only partly, based on limited knowledge and understanding of its processes. Although it is possible, to describe the system using necessary number of equations, these equations are often either not complete, or contain unknown parameters and constants.

In this situation, it is necessary to conduct additional measurements, to gather adequate amount of data of sufficient quality, that will be used to identify the unknown parameters and constants of the description. Only after experimentally acquiring these unknown parameters is the description complete.

- **Black box** system identification assumes, that the identified system can not be defined at all, due to very limited or nonexistent knowledge and understanding of its processes. Therefore, it is not possible to describe it with equations.

  In this case, the only way to understand the system, are extensive experimental measurements followed by thorough examination of the acquired data. By observing how the system transforms the inputs into outputs, it is often possible to recognize certain patterns. After proposing the possible structure of the model based on those patterns, the gray box approach can be applied, where the unknown parameters are being identified based on data. If the fit between the model and data is not sufficient, the proposed structure is modified or replaced with another altogether, and the process is repeated until the fit is sufficiently accurate. Black box identification then does not provide us with a model describing the system, but rather a model describing the data that it was created with—the structure of such a model is not based in physical principles, but on the fit accuracy it is able to achieve.

Based on the three aforementioned identification approaches, the beginning statement about the inseparability of modeling and identification can be completed—it could be even said, that they can be considered being a single process, as the modeling serves for finding a mathematical description of the system, and the identification serves for revealing the unknown parameters in that description.

To summarise, while in the white box approach it is possible to outright create a complete model of the system, in the gray box identification approach this model has to be completed by identification of its parameters through data, and the black box approach utterly relies on the identification from data—as there is no predefined structure of the model. The process of modeling and identification is then used to acquire a complete mathematical model of the specific system.

In our case, the gray box approach is used—while mathematical models that describe the FloatShield system with sufficient accuracy have been created, some of their parameters are still not known.

More specifically, the parameters K_f, τ_1, τ_2 in the linearized transfer function (4.11) and also in the linearized state-space model (4.16) are not known, while the parameters α, K_f, τ_2 in the nonlinear (quadratic) state-space model (4.17) are not known. Therefore, all of them have to be found based on the data acquired from the operation of the FloatShield system.

To be able to acquire correct identification data a specialized system input signal had to be created. "Correct" in a sense, that the data will represent the characteristic behaviour of the system—in our case, the response of the system in steady state to the input change.

In [7] the use of pseudo-random binary sequence (PRBS) signal for system identification was proposed, while in [10] the authors argued that amplitude modulated pseudo-random binary sequence (APRBS) signal would be more suitable for a nonlinear system.

The PRBS signal has a form of a square wave, that alternates between two fixed values with varying signal durations, while APRBS signal additionally pseudo-randomly changes the amplitude within the range of two fixed values. The term "pseudo-random" is used here, as although these signals look completely random, they are replicable—they are created by the process that for the same inputs produces the same outputs—and thus are not random in the true sense.

Both signals have been created within the MATLAB software environment by the scripts `prbsGenerate` and `aprbsGenerate` as a part of the AutomationShield library. Their source code can be found in appendices B.8 and B.9. These two scripts create the respective signals in range from -1 up to 1, meaning that the PRBS signal alternates strictly between these two values and APRBS signal changes pseudo-randomly within the limits set by these two numbers. Both types of input signals have been tested, and at the end, usable identification data were obtained by utilizing the PRBS signal.

In order to create the final input signal, the constant values of the system input at equilibrium, and of the maximal change have to be defined. The final APRBS signal is generated, by adding the value of maximal change, multiplied by the generated pseudo-random signal, to the value of system input at equilibrium.

The very same procedure, was used to generate the PRBS signal, that is, together with the corresponding system response depicted in Fig. 4.1.

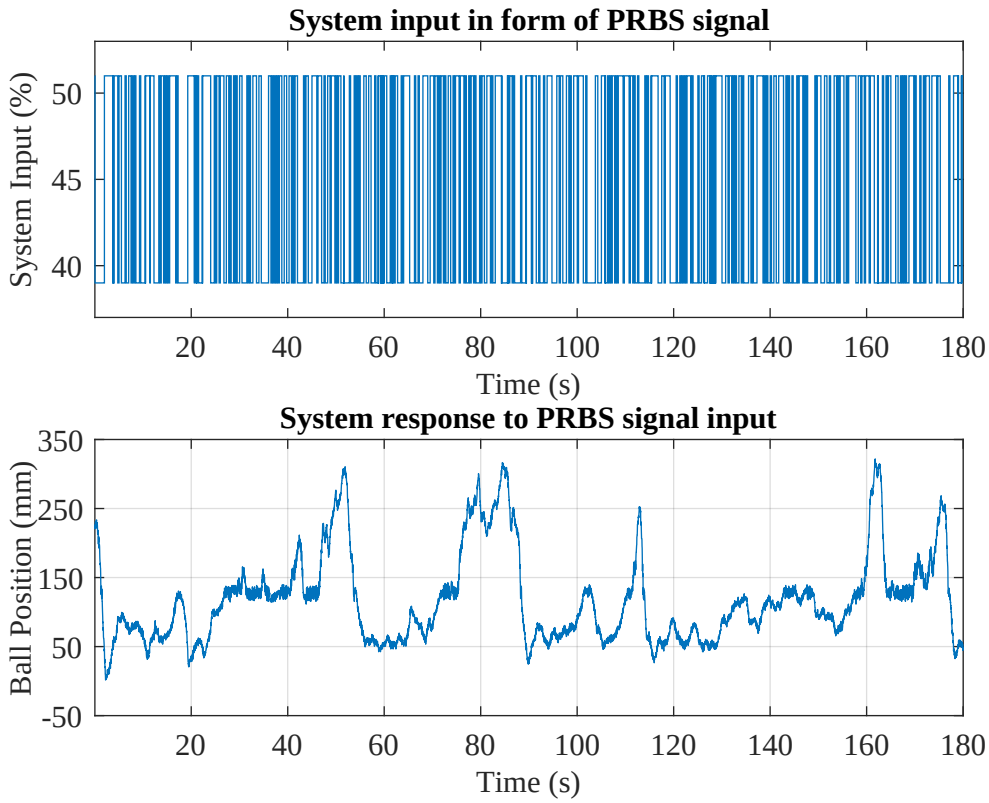


Figure 4.1: System response to the identification signal.

Acquisition of respective data was done through `FloatShield_Identification` Arduino IDE sketch provided in appendix A.6.

Using such a large amount of data for model fitting would be impractical, therefore only the part was extracted, that was deemed stable and characteristic of the system. This part is shown in Fig. 4.2

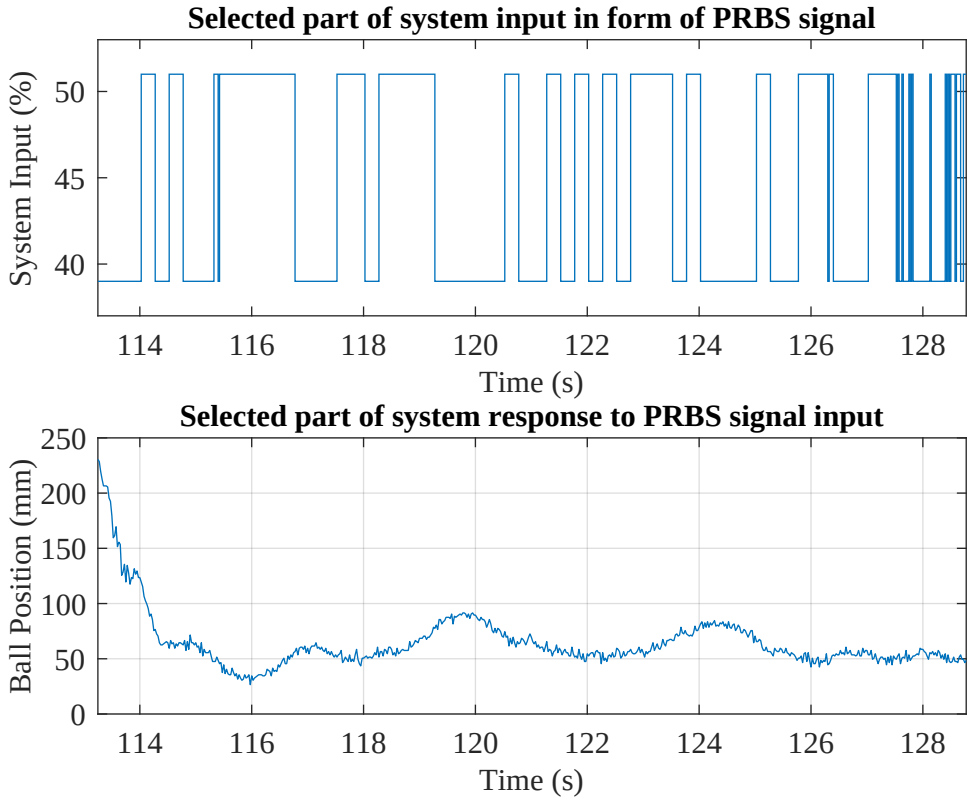


Figure 4.2: Chosen segment of the system response to the identification signal.

The complete process of the gray box identification of the three FloatShield models is included in the MATLAB script within the AutomationShield library under the name `FloatShield_Ident_Greybox`, provided in appendix B.7. Let us shortly describe this script.

At first, the identification data are loaded from the MATLAB file, where they are stored under the name `resultID`, and are consequently saved into a MATLAB identification object. This object stores all the necessary information about the data, such as the sampling period used to acquire the data, or the units of the individual measured values.

Then, a specific segment is selected from within the data, that will be actually used for identification. Subsequently, this segment is detrended, what in simple terms means shifting the data so that their mean value becomes near zero, and filtering them to remove their linear trends.

This is followed by the specification of known system parameters such as the weight of the ball, its diameter, or the density of air, and by providing approximate values of unknown parameters—that will be used as initial values when searching for a better fit.

After that, the procedure becomes specific for each individual model—at this point the user has to choose which of the three models should be identified.

Generally, parameters of the system that are known and therefore constant can be specified, and also the ones that are unknown and therefore should be searched for. Additionally, parameters such as the maximal number of iterations can be set (within which should the MATLAB solver find the best fit), type of the solver, or focus of the solver—whether our priority is that the model will be stable, or that it will achieve better fit with the provided data.

Finally, the model can be saved to an external MATLAB file, from which it can be imported into other MATLAB scripts and utilized further.

A comparison of match achieved by individual models, is shown in Fig. 4.3.

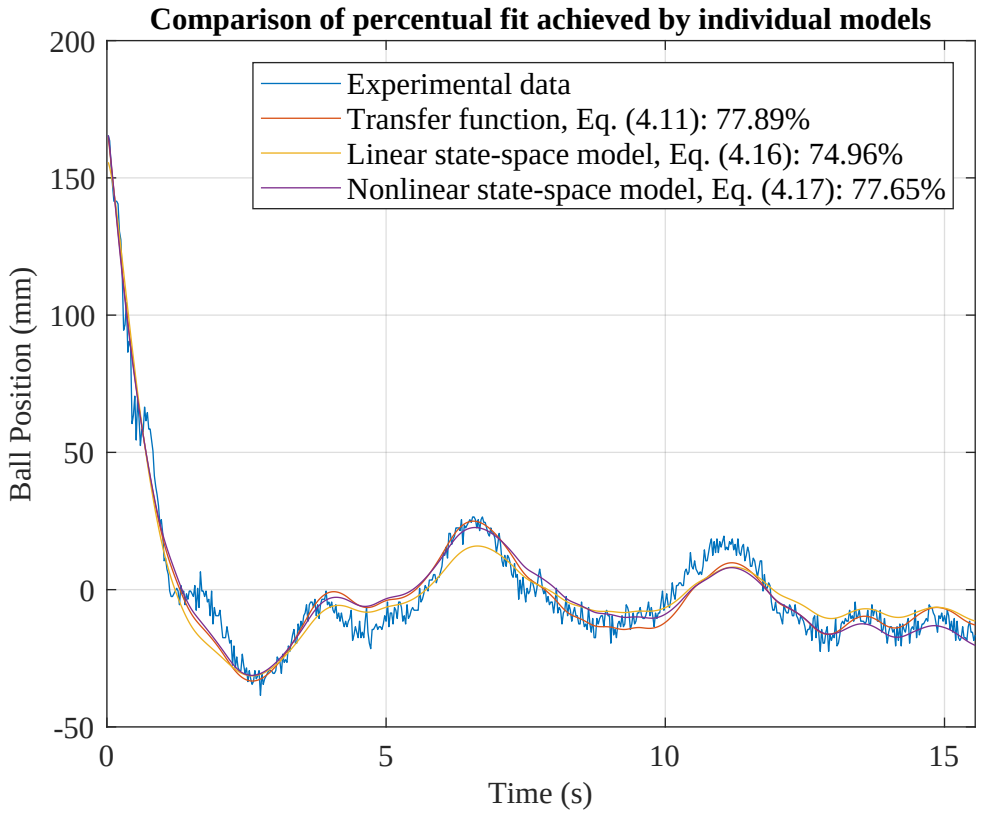


Figure 4.3: Comparison of models to data fit.

As it can be seen in Fig. 4.3—where the outputs calculated through models are compared with the original identification data—all three models provide a fairly accurate approximation of the system.

Based on the data shown in Fig. 4.2, all the unknown parameters have been identified, and thus have been obtained three usable mathematical models of the FloatShield device.

4.3 Estimation

The process of estimation, does not directly relate to the processes of modeling and identification, but is dependent on them, as it requires an accurate mathematical model.

Note, that under the term of estimation, Kalman filter estimation is meant here.

Certain higher forms of system control are based on the knowledge of state-space representation of the system, and all of the system states. So far, the linear (linearised) state-space representation has been acquired (4.16), but two of the three system states that were previously defined within the state vector as $\begin{bmatrix} h_{(t)} & \dot{h}_{(t)} & v_{(t)} \end{bmatrix}^{\mathrm{T}}$, are unknown. Although the FloatShield device is equipped with a TOF sensor that provides measurements of the first state, there are no additional sensors.

The second state—ball velocity—may be calculated from the position measurements by using backward difference approximation of first derivation. However, the position measurements are often very noisy, due to the fact that the TOF distance sensor operates in the high speed mode, where the speed is on the expense of accuracy. Because of that, the difference between the two adjacent measurements would never be sufficiently accurate.

When it comes to the third state—air velocity—there is currently no method at our disposal, to measure or approximate it. There is no other way of obtaining it, than through a Kalman filter estimation.

The basic process of estimation through the Kalman filter algorithm, was previously mentioned through the `getKalmanEstimate()` Arduino IDE method in Chap. 3.3.4, and by the `estimateKalmanState()` MATLAB function in Chap. 3.4.3. Let us now look at the algorithm itself:

$$\hat{\mathbf{x}}_{(k)}^{-} = \mathbf{A}\hat{\mathbf{x}}_{(k-1)} + \mathbf{B}\mathbf{u}_{(k)}, \tag{4.18a}$$

$$\mathbf{P}_{\mathbf{K}_{(k)}^{-}} = \mathbf{A}\mathbf{P}_{\mathbf{K}(k-1)}\mathbf{A}^{\mathrm{T}} + \mathbf{Q}_{\mathbf{K}}, \tag{4.18b}$$

$$\mathbf{K}_{(k)} = \mathbf{P}_{\mathbf{K}_{(k)}^{-}}\mathbf{C}^{\mathrm{T}}\left(\mathbf{C}\mathbf{P}_{\mathbf{K}_{(k)}^{-}}\mathbf{C}^{\mathrm{T}} + \mathbf{R}_{\mathbf{K}}\right)^{-1}, \tag{4.18c}$$

$$\hat{\mathbf{x}}_{(k)} = \hat{\mathbf{x}}_{(k)}^{-} + \mathbf{K}_{(k)}\left(\mathbf{y}_{(k)} - \mathbf{C}\hat{\mathbf{x}}_{(k)}^{-}\right), \tag{4.18d}$$

$$\mathbf{P}_{\mathbf{K}(k)} = \left(\mathbf{I} - \mathbf{K}_{(k)}\mathbf{C}\right)\mathbf{P}_{\mathbf{K}_{(k)}^{-}}, \tag{4.18e}$$

where:

- $\mathbf{A}$ is the system matrix of a discrete state-space model;
- $\mathbf{B}$ is the input matrix of a discrete state-space model;
- $\mathbf{C}$ is the output matrix of a discrete state-space model;
- $\mathbf{I}$ is the identity matrix;
- $\mathbf{Q}_{\mathbf{K}}$ is the process noise covariance matrix;
- $\mathbf{R}_{\mathbf{K}}$ is the measurement noise covariance matrix;
- $\mathbf{P}_{\mathbf{K}_{(k)}^{-}}$ is the a priori estimation error covariance matrix at discrete time k;

- $\mathbf{P}_{\mathbf{K}(k)}$ is the a posteriori estimation error covariance matrix at discrete time k;
- $\mathbf{K}_{(k)}$ is the Kalman gain matrix at discrete time k;
- $\hat{\mathbf{x}}^{-}_{(k)}$ is the a priori state vector estimate at discrete time k;
- $\hat{\mathbf{x}}_{(k)}$ is the a posteriori state vector estimate at discrete time k;
- $\mathbf{u}_{(k)}$ is the system input vector at discrete time k;
- $\mathbf{y}_{(k)}$ is the system output vector at discrete time k.

First, an a priori state estimate $\hat{\mathbf{x}}^{-}_{(k)}$ is calculated based on the system state-space matrices $\mathbf{A}$, $\mathbf{B}$, state estimate from previous iteration $\hat{\mathbf{x}}_{(k-1)}$, and the current system input $\mathbf{u}_{(k)}$ in (4.18a).

Although it has not been explicitly stated, it is easily discernible that the FloatShield system is a so-called single-input single-output (SISO) system—meaning that only one parameter is provided to the system and consequently only one parameter is returned. Therefore, the system input vector $\mathbf{u}_{(k)}$ and system output vector $\mathbf{y}_{(k)}$ in the case of FloatShield, turn out to be scalar values $u_{(k)}$, $y_{(k)}$.

Second, an a priori estimation error covariance matrix $\mathbf{P}_{\mathbf{K}^{-}_{(k)}}$ is calculated based on the system state-space matrix $\mathbf{A}$, the estimation error covariance matrix from the previous iteration $\mathbf{P}_{\mathbf{K}(k-1)}$, and the process noise covariance matrix $\mathbf{Q}_{\mathbf{K}}$ in (4.18b).

In the third step (4.18c), the Kalman gain vector $\mathbf{K}_{(k)}$ is calculated based on the system state-space matrix $\mathbf{C}$, the a priori estimation error covariance matrix $\mathbf{P}_{\mathbf{K}^{-}_{(k)}}$, and the measurement noise covariance matrix $\mathbf{R}_{\mathbf{K}}$.

Subsequently, the a posteriori state estimate $\hat{\mathbf{x}}_{(k)}$ is calculated by using the difference between the actual system output measurement $\mathbf{y}_{(k)}$ and the output a priori estimate $\mathbf{C}\hat{\mathbf{x}}^{-}_{(k)}$ to update the a priori state vector estimate $\hat{\mathbf{x}}^{-}_{(k)}$ in (4.18d).

Finally, the a posteriori estimation error covariance matrix $\mathbf{P}_{\mathbf{K}(k)}$ is calculated by using the state-space matrix $\mathbf{C}$, identity matrix $\mathbf{I}$, Kalman gain vector $\mathbf{K}_{(k)}$ and the a priori estimation error covariance matrix $\mathbf{P}_{\mathbf{K}^{-}_{(k)}}$ in (4.18e).

The process, described by Eqs. (4.18), has to be repeated in every sample.

It can be said, that the Kalman filter utilizes a two-step predictor-corrector algorithm, where the equations (4.18a) and (4.18b) represent the prediction phase, and the equations (4.18c) to (4.18e) represent the correction phase.

The enormous contribution of the Kalman filter algorithm (4.18), is visually shown in the `FloatShield_KalmanFiltering` MATLAB script file provided in appendix B.10, where this algorithm is used with the identified linear state-space model (4.16), on the data that were used in identification (shown in Fig. 4.2). The algorithm is implemented in the `estimateKalmanState()` MATLAB function.

The results acquired through that script are depicted in Fig. 4.4, where all of the three FloatShield system states are being estimated.

Note, that the estimated results shown in Fig. 4.2 should be interpreted as changes from the steady-state of the system—because the identification data were detrended before the identification procedure, and also, due to the perturbational form of the linear state-space model (4.16).

The accuracy of the estimation largely depends on the quality of the state-space model, but it is also possible to significantly impact the results by setting specific values of $\mathbf{Q_K}$, $\mathbf{R_K}$ covariance matrices. The `FloatShield_KalmanFiltering` MATLAB script was originally intended to be used as a tool for finding an ideal combination of covariance matrices, and it enables the user to see what effects the changes have on the estimated states. Also, in our case, the estimates shown in Fig. 4.4 were obtained only after a careful hand-tuning of the covariance matrices.

All three states—ball position, ball velocity, air velocity—are shown on the charts of Fig. 4.4 in a descending order.

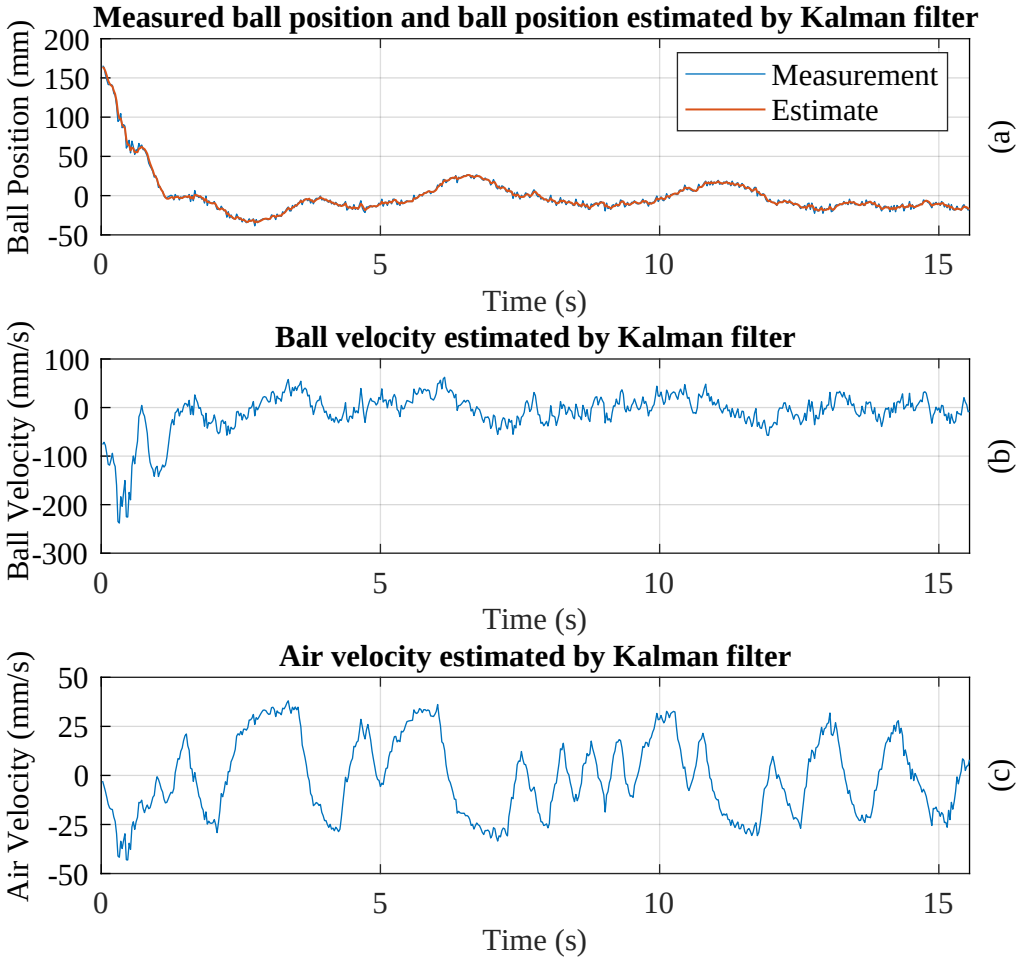


Figure 4.4: System states estimated using Kalman filter on identification data.

As it is clear from Fig. 4.4(a), the ball position estimate corresponds to the actual position measurement very closely, where a slight effect of filtering can also be noticed, as the estimate curve is smoother. However, filtering was not the goal.

On the second graph in Fig. 4.4(b), the estimated velocity of the ball can be seen, which in our case of limited equipment, could be validated only through using the difference approximation of first derivation. The validation is depicted in Fig. 4.5.

It has been previously stated, that the differentiation of the fairly noisy measurement of the position, could yield only inaccurate velocity readings.

This is confirmed in Fig. 4.5, where the blue curve shows the velocity that was calculated from the position using the backward difference approximation of first derivation $\dot{h}_{(k)} \approx \frac{h_{(k)} - h_{(k-1)}}{T_s}$, where T_s is the sampling period.

On the other hand, the orange curve in Fig. 4.5, shows the very same method of velocity calculation, but in this case, the velocity is calculated from the slightly filtered measurement (estimate of the first state) as shown in Fig. 4.4(a). As it can be noticed, even the visually insignificant difference between the estimate and the measurement depicted in Fig. 4.4(a), has a major impact on the ball velocity calculated from it.

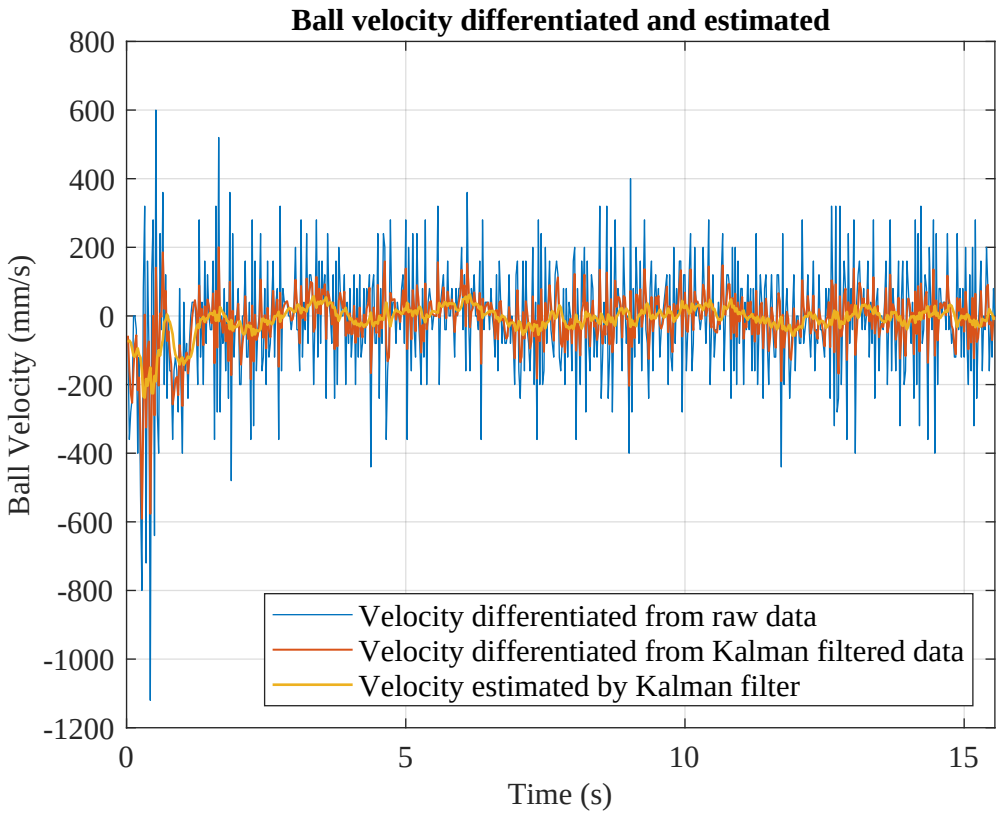


Figure 4.5: Validation of second state estimate based on differentiation of measured data.

Finally, the yellow curve represents the ball velocity estimated as a second system state by the Kalman filter algorithm in Fig. 4.5—the same as is shown in Fig. 4.4(b). When compared to the other two curves, the estimate looks fairly reasonable, thus it will be further assumed to be the true velocity of the ball.

However, when it comes to the third state estimate—the air velocity depicted in Fig. 4.4(c)—its validity can not be confirmed by experiment, as there are no sensors that would enable us to measure the air velocity within the tube present on the FloatShield device or at our disposal.

Due to the facts, that there is no way of validating the last estimate, and that the air velocity values stay within reasonable limits while the ball velocity values appear to be correct, this estimate can be (for our purposes) assumed to be the true velocity of the air inside the tube.

Therefore, the three forms of FloatShield system model (based on Newton's second law of motion) have been created, and their unknown parameters have been identified. It was confirmed, that (at least in the case of linearized state-space model) by using this model, very reasonable estimates of the system states can be acquired—meaning that it sufficiently describes the FloatShield system.

One last thing that has to be mentioned is, that due to the fact the models were obtained from continuous equations, they are also continuous. To be able to work with them in iterative algorithms, they have to be discretized. A MATLAB native function under the name `c2d()` is being used for that purpose, which, discretizes the continuous model based on the provided input arguments.

If using the `c2d()` function is not an option, the discrete state-space system matrices $\mathbf{A}$ and $\mathbf{B}$ can be obtained from their continuous counterparts through the following expressions

$$\mathbf{A} = e^{\mathbf{F_c} T_s}, \tag{4.19a}$$

$$\mathbf{B} = \mathbf{F_c}^{-1} \left(\mathbf{A} - \mathbf{I}\right) \mathbf{G_c}, \tag{4.19b}$$

where:

- $\mathbf{F_c}$ is the system matrix of a continuous state-space model;
- $\mathbf{G_c}$ is the input matrix of a continuous state-space model;
- T_s is the sampling period;
- e is the Euler's number, and the operation is matrix exponential.

Alternatively, if for any reason they can not be calculated directly by using Eq. (4.19) or `c2d()` function, they can be approximated with the expressions

$$\mathbf{A} \approx \mathbf{I} + \mathbf{F_c} T_s, \tag{4.20a}$$

$$\mathbf{B} \approx \mathbf{G_c} T_s. \tag{4.20b}$$

Chapter 5

Feedback Control

When it comes to dynamic systems, the main reason for learning about them, trying to understand them and mathematically describing them is to control them effectively.

The ability to direct and influence the behaviour of a dynamic system is usually meant under the term control. This is most often realised—as every system can be characterized through the relationship between its inputs and outputs—by the so-called controllers or control algorithms, which calculate such a system input that will influence the output to get closer to the reference value.

There are many types of control algorithms, but most often the controller forms a closed loop in the system, with the objective of calculating the required system input. In order to achieve that, it uses the current value of system output measurement, and also the reference value to which the output will be compared. The reference, also known as the set point, represents the information provided to the controller, that specifies the manipulated system parameters to which the control algorithm shall drive the dynamic system.

Here, the three forms of control are described, that will be implemented on the FloatShield device as a part of this thesis. Specifically, the FloatShield system shall be controlled through the use of:

- PID controller,
- LQ regulator,
- MPC.

Let us examine each one of these means of control through examples.

5.1 Structure of Examples

In the following chapters, sample experiments implementing the specific form of control for each software platform will be provided, after briefly explaining the necessary theory behind the individual forms.

To avoid unnecessary repetitiveness, the default structure of the Arduino IDE and MATLAB experiments are outlined here, and the respective chapters list only the deviations from this default.

In the case of Simulink examples, the default structure can not be as easily defined, and due to their visual nature it will be best to describe them individually.

Each experiment can be conceptually divided into four distinct phases, they are the:

- **Preparation** phase, which in the case of Arduino IDE examples consists of including the `FloatShield.h` API, that contains all the functions available for the FloatShield device, and the `Sampling.h` header file as a part of the AutomationShield library, providing the user with functions realising a dependable method of sampling on the Arduino boards. Optionally, a choice can be made, whether the reference will be automatically provided from the predefined reference vector, or will be supplied manually, using the potentiometer runner of the device.

  In the case of MATLAB examples, this phase consists of clearing the *Workspace* variables left from the previous executions and creating the `FloatShield` object of the `FloatShield` class. This object has to be initialised manually, because in MATLAB FloatShield API (as opposed to its Arduino IDE counterpart), it is not done so by default.

- **Initialisation** phase, which in the case of Arduino IDE examples consists of defining important constants and variables such as values of the sampling period `Ts` representing the time between samples, a reference trajectory sequence `R[]` containing individual references, and section length `T` that specifies for how many samples should each value from the reference trajectory be used as a reference for the controller. This is followed by the start of serial communication (to be able to send data back to the computer), initialisation and calibration of the device through `begin()` and `calibrate()` methods described in Chap. 3.3.2. Finally, the `Sampling` object of the `Sampling.h` library is initialised with the specified sampling period `Ts`.

  In the case of MATLAB examples, this phase also consists of defining important constants and variables such as the sampling period `Ts`, reference trajectory sequence `R`, and section length `T`. This is followed by the initialisation and calibration of the device through `begin()` and `calibrate()` methods described in Chap. 3.4.2. Sampling in MATLAB is realised through the `tic-toc` functions that are able to measure the time elapsed in between their individual calls.

- **Control algorithm** which in both cases of examples starts with a short takeoff routine that ensures the ball takes off of the base before starting the measurements—as depending on the controller tuning, it can take a particularly long time.

  What follows afterwards, is characteristic of each individual example and it will described locally in each respective chapter.

- **Data acquisition** which in both cases of examples consists of saving the results in a form of a three column table, where the individual columns represent the values of reference `r`, system output `y` and system input `u`, while the rows represent the respective samples.

  In the case of the Arduino IDE examples, the results are sent through serial communication and are stored using e.g. CoolTerm serial port terminal application, in a simple text file (.txt suffix).

In the case of MATLAB examples, the results are being stored in the `response` matrix, that is saved at the end of the experiment, under an example-specific name as a MATLAB file (.mat suffix).

Note, that until it is explicitly stated otherwise, the following results were obtained with the FloatShield R1 device mounted on the Arduino UNO board, using a 25 ms sampling period `Ts`.

5.2 PID Controller

The PID controller is possibly the most well known while also one of the simplest ways of controlling the system.

Its popularity stems from its universal applicability and also from the fact, that it does not depend on the knowledge of the system—it can be implemented even on systems that are not, or for any reason can not be mathematically described. It operates solely using the outputs of the system and the provided reference value, from which it calculates the required system input and provides it to the system in a closed loop.

One of the disadvantages of PID controllers is, that they are applicable only on SISO systems, what may in some cases prevent their effective application.

If the PID controller, for any reason, had to be used to regulate processes in multiple-input multiple-output (MIMO) systems, the MIMO system would first need to be transformed into a composition of SISO systems by a method of decoupling (if possible), and then, the PID controllers would have to be designed for the resulting SISO systems individually.

The PID controller in its standard form is defined by the following control law

$$u_{(t)} = K_p \left(e_{(t)} + \frac{1}{T_i} \int_0^t e_{(\tau)} d\tau + T_d \frac{de_{(t)}}{dt} \right), \tag{5.1}$$

where:

- $u_{(t)}$ is the calculated system input at time t;
- $e_{(t)}$ is the error between reference and system output at time t;
- K_p is the proportional gain;
- T_i is the integral time;
- T_d is the derivative time.

As can be seen in Eq. (5.1), the PID controller takes the current value of error (calculated by subtracting the current system output from the current reference), integral of this error over time and derivative of this error in time. Then, the controller evaluates the influence of each individual form of error on the final system input, based on the respective constants.

Note, that the current reference represents the value to which the controller should drive the output to, in the current time step.

While being relatively simple, the PID controller utilizes a quite intelligent approach—it takes not only the current value of error into account, but also the value of its integral, which can be thought of as a representation of the errors from the past; and the value of its derivative, which can be thought of as a prediction of the future error. This enables it to calculate the system input based on not only the present, but also the past and the near future.

The impact of the individual forms of the error depends on the values of the K_p, T_i and T_d constants, that have to be carefully selected as they are the defining aspect of the PID controller—they heavily influence the accuracy and stability of control.

There are many quantitative approaches of tuning these constants, however, it is possible to find a sufficiently accurate combination by trial and error, while the time it takes to find better constants decreases with experience—as the effect caused by the change of individual constants is not random, but rather predictable.

To be able to use the PID controller in iterative examples, it has to be transformed into a discrete form—specifically the integral in time will be replaced by a sum up to the current step, and the derivative will be replaced by a backward difference approximation of derivation.

The control law in its standard discrete form will then be

$$u_{(k)} = K_p \left(e_{(k)} + \frac{1}{T_i} \sum_{j=1}^{k} e_{(j)} + T_d \frac{e_{(k)} - e_{(k-1)}}{T_s} \right), \tag{5.2}$$

where T_s is the sampling period of the controlled discrete system.

Another disadvantage of the PID controller is, that it does not take into consideration the process limits of the system input—for example, while the system is able to accept only a percentual value from range of 0%–100% as an input, the PID controller can easily calculate values out of this range, whether positive or negative. This deficiency is in practice then "solved" by programmatically constraining the input into an acceptable range, before providing it to the system. However, this renders the controller non-linear, and thus affects stability.

5.2.1 PID Example in Arduino IDE

An implementation of the PID control algorithm represented by Eq. (5.2) is realised within the Arduino IDE in a form of a sketch by the name `FloatShield_PID` (.ino suffix). Its content is provided in appendix A.7. This file, with all the other sketches, scripts and models, can be found within the examples folders of the AutomationShield library. The example adheres to the structure described in Chap. 5.1, with the following additions and differences:

The **Initialisation** phase is characterized by the definition of the reference array `R[]` and the section length `T` as

```
float R[] ={65,50,35,45,60,75,55,40,20,30};// Trajectory in %
int T = 2400;                               // Section length
```

followed by the definition of the constants for PID controller, using a preprocessor directive.

```
#define KP 0.25                                    // PID Kp constant
#define TI 5                                       // PID Ti constant
#define TD 0.01                                    // PID Td constant
```

These constants—obtained by manual tuning—are then together with sampling period `Ts` provided to the `PIDAbs` object from the `PIDAbs.h` header file, that is natively a part of the AutomationShield library. This object includes the method utilizing the PID control law (5.2), and the input will be calculated through it.

The main **Control algorithm** (after the short takeoff routine) then consists of

```
r = R[i];                                          // Select reference
y = FloatShield.sensorRead();                      // Read sensor
u = PIDAbs.compute(r-y,0,100,0,100);               // Calculate PID
FloatShield.actuatorWrite(u);                      // Actuate
```

where first a reference `r` is chosen from the array `R[]` based on the number of samples that have already passed, and the measurement of the ball position (altitude) in percentages `y` is taken. Then, the system input `u` is calculated through the `PIDAbs` object, based on the difference of `(r-y)` representing the error. Note, that the `PIDAbs` object additionally offers the option of directly constraining the values of calculated input and of the integration part of the PID equation (so-called integral windup). In this case both are constrained to adhere to a percentual range. Finally, the input `u` is fed to the corresponding FloatShield method providing it to the device fan.

Figure 5.1 depicts the results obtained by the `FloatShield_PID` (.ino) example.

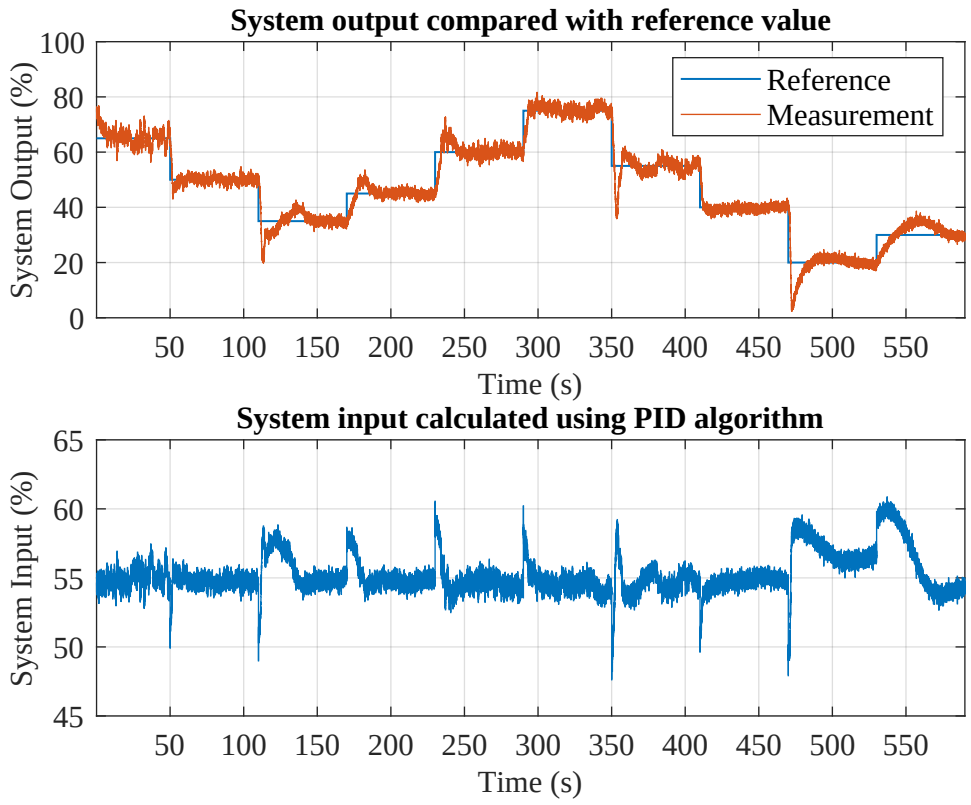


Figure 5.1: Results of the FloatShield PID Arduino IDE example.

The values of the measurements (orange) can be seen on the upper figure, with reference values (blue) in background; while the lower figure depicts the system input calculated by the PID algorithm. The parts where the reference suddenly changes and the controller has to react to the change are most notable. Besides those parts, the system input is being held at an approximately constant 55% of the power, what can be thought of as a power level corresponding to steady state. The reference is being tracked fairly accurately, with occasional overshoots after sudden reference change.

Due to the combination of sampling period `Ts`, section length `T` and the length of the reference array `R[]`, the experiment took slightly more than 10 minutes (including calibration and takeoff).

5.2.2 PID Example in MATLAB

The PID control algorithm defined by Eq. (5.2) is implemented within the MATLAB software environment in the script named `FloatShield_PID` (.m suffix). Its code can be found in appendix B.11. The example respects the structure described in Chap. 5.1, with the following differences:

The **Preparation** phase is extended by the definition of the `PID` object of the `PID` class

```
PID = PID;                                  % Create PID object
```

where the `PID` class was created as a MATLAB counterpart of the `PIDAbs.h` Arduino IDE class, and it provides the user with the same functionality.

Next, the reference vector `R` and the section length `T` are defined in the **Initialisation** phase as

```
R = [65,50,35,45,60,75,55,40,20,30]; % Trajectory in %
T = 2400;                                   % Section length
```

followed by the definition of constants for the PID controller as separate variables.

```
Kp = 0.25;                                  % PID Kp constant
Ti = 5;                                     % PID Ti constant
Td = 0.01;                                  % PID Td constant
```

Analogically to the Arduino IDE, these constants are then, together with the sampling period `Ts`, provided to the `PID` object of the `PID` class. The `PID` object includes the method implementing the PID law (5.2), through which the output is calculated.

Then, the main **Control algorithm** includes

```
r = R(i);                                   % Select reference
y = FloatShield.sensorRead();               % Read sensor
u = PID.compute(r-y,0,100,0,100);           % Calculate PID
FloatShield.actuatorWrite(u);               % Actuate
```

where first a reference `r` is picked from the vector `R` based on the number of elapsed samples. Then, the measurement of the ball position (altitude) in percentages `y` is taken, and the system input `u` is calculated through the `PID` object, based on the difference of `(r-y)` representing the error. Similarly as in the Arduino IDE, the `PID` object offers the option of constraining the values of calculated input and of the error sum. Both of

them are constrained to the percentual range. Finally, the calculated input `u` is fed to the corresponding FloatShield method that provides it to the fan of the device.

A plot of the results acquired by the `FloatShield_PID` (.m suffix) script is shown in Fig. 5.2.

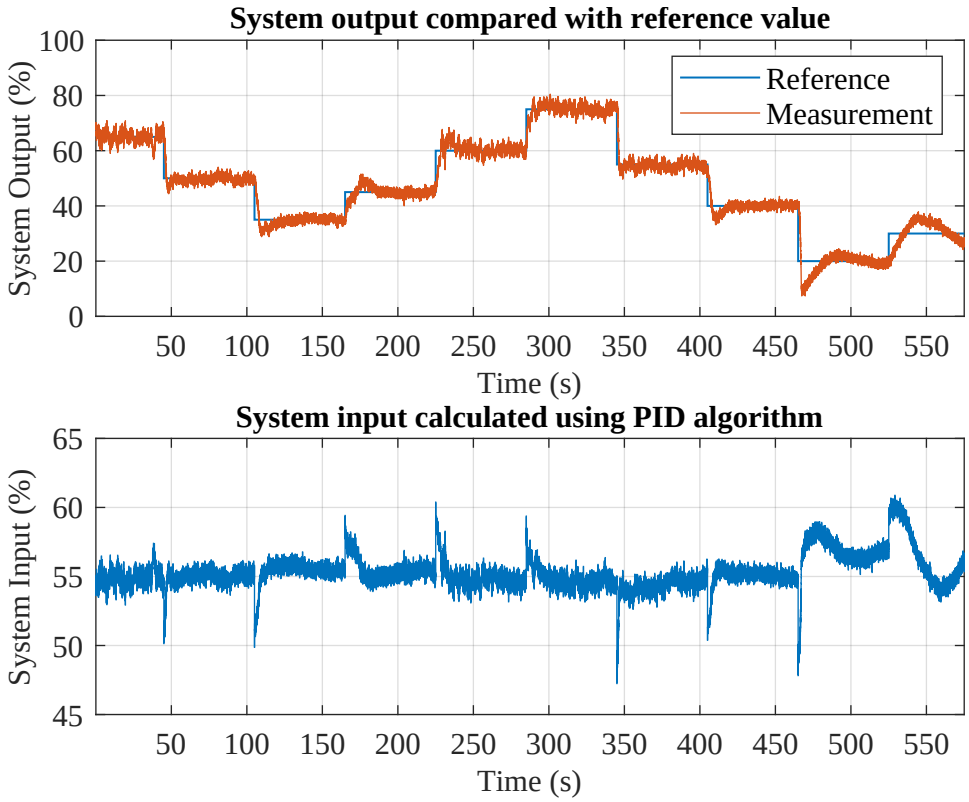


Figure 5.2: Results of the FloatShield PID MATLAB example.

These results are almost identical to the ones obtained by the Arduino IDE PID example. In this case, there have been slightly fewer overshoots after reference change, than in the Arduino IDE counterpart, while the reference tracking remained reasonably accurate. This is not very surprising, as the same controller parameters were used. As before, the example took approximately 10 minutes.

Note, that while the results from Arduino IDE shown in Fig. 5.1 and from MATLAB shown in Fig. 5.2 look very similar, due to the fact that MATLAB utilizes only an imperfect form of sampling through `tic-toc` functions, and as the communication between the MATLAB and Arduino is limited by the USB connection, the results acquired from the MATLAB example have only a limited scientific value.

5.2.3 PID Example in Simulink

Within the Simulink environment, the FloatShield PID control example can be accessed through the model named `FloatShield_PID` (.slx suffix). As it was already explained in Chap. 3.5.1, the lines of code are in Simulink replaced by blocks. The Simulink PID example can be therefore represented by a single picture—by Fig. 5.3

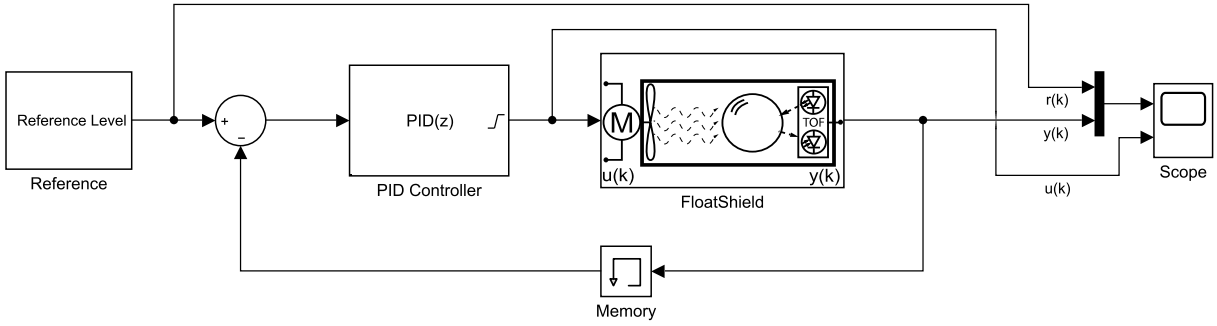


Figure 5.3: The Simulink model representing the FloatShield PID example.

where the individual blocks were introduced and explained in Chap. 3.5.

One of the advantages of Simulink is, that it quite literally shows the flow of information—represented by the connecting lines between the blocks. This, of course, might become a disadvantage in the case that there are many blocks in the model and lines overlap. However, in our case it only brings more visual clarity to the example.

Let us review the flow of information in the Simulink FloatShield PID example. First, the current reference from the `Reference` block is led to the summation block, where the system output is subtracted from it—thus the output of the summation block will be the error.

Second, the error from the summation block is supplied to the `PID Controller` block—a native Simulink block that has the PID constants specified within its settings—which then uses the current error to calculate the system input.

Afterwards, system input is calculated and provided to the FloatShield system, represented in the model by the `FloatShield` block, where the input is sent to the system actuator (fan).

Finally, the measurement of the ball position is led back to the summation block as an output from the system, to calculate the error for the PID controller. The `Memory` block along this path serves here as a one step delay for the measurement, as first there has to be an input and only then an output—to prevent algebraic loops.

Reference, system input and system output are provided to the `Scope` block throughout the simulation, acting as real-time plotter of the data and at the same time it is continually saving the results into the `SimulinkPIDResponse` MATLAB file (.mat suffix).

Note, that the process of calibration is realised at the beginning of the experiment through the `FloatShield` block.

The results from the `FloatShield_PID` (.slx suffix) Simulink example, are illustrated in Fig. 5.4 on the following page.

In contrast to its Arduino IDE and MATLAB counterparts, the system input was not calculated through our own function in this example—a native Simulink PID block was utilised for this purpose. That might be the probable reason for the input curve being shaped differently in comparison to the previous results. Additionally, the system input is held at a slightly lower level than before—at approximately 53% of power. The reference tracking is in this case still sufficiently accurate, although with more noticeable overshoots after sudden reference change. This example took approximately 10 minutes as well.

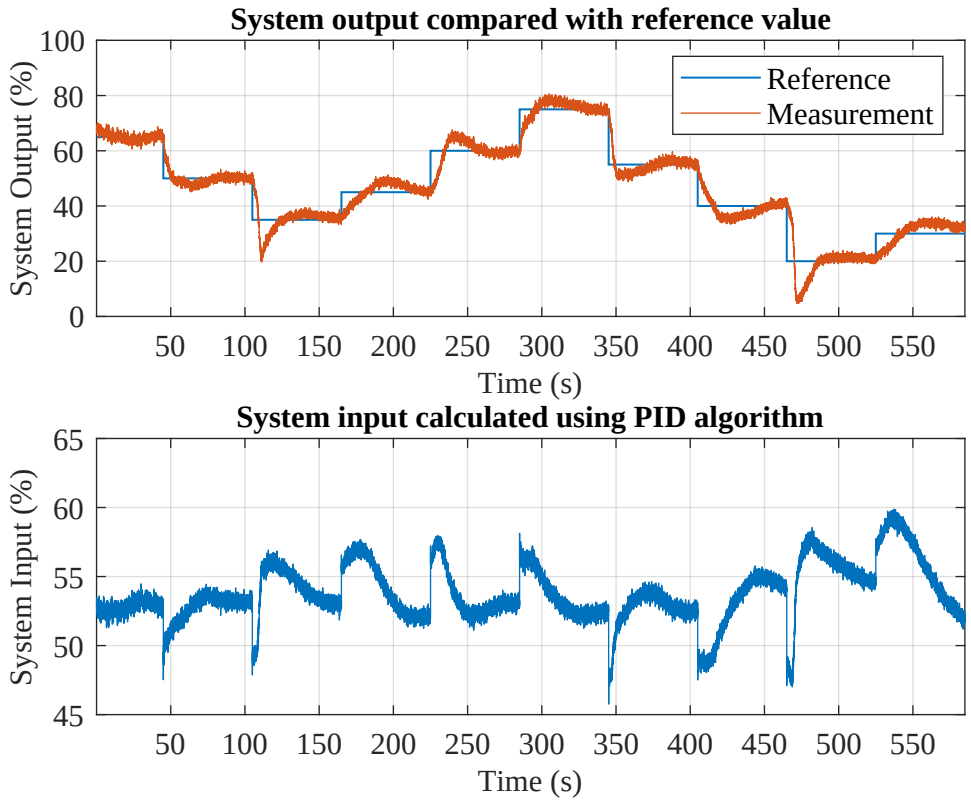


Figure 5.4: Results of the FloatShield PID Simulink example.

Note, that the Simulink does not suffer from the irregular sampling and slow communication problems as MATLAB does—the results acquired from Simulink examples can be considered real-time.

5.3 LQ Regulator

The linear quadratic regulator (LQR) is a more complex form of control than the PID, as it relies on the knowledge of the mathematical model of the controlled system—specifically the state-space model.

While LQR still shares the disadvantage of PID, where the controller calculates system input regardless of the system process limitations (so the input has to be constrained before being provided to the system), it is not limited in any way when it comes to being applied on MIMO systems. It is, however, necessary to say, that despite not taking into consideration process limits, the input calculated by the LQR has a more sophisticated fundamental logic behind it.

The level of sophistication might not be initially obvious from the LQR control law, which is defined as

$$\mathbf{u}_{(k)} = -\mathbf{K}\mathbf{e}_{(k)} = -\mathbf{K}\left(\mathbf{x}_{(k)} - \mathbf{x}_{r(k)}\right), \tag{5.3}$$

where:

- $\mathbf{u}_{(k)}$ is the calculated system input vector at discrete time k;
- $\mathbf{K}$ is the gain matrix;
- $\mathbf{x}_{(k)}$ is the system state vector at discrete time k;
- $\mathbf{x}_{r(k)}$ is the reference state vector at discrete time k;
- $\mathbf{e}_{(k)}$ is the state error with respect to the reference at discrete time k.

But with a closer look, it can be noticed that instead of a simple scalar error value, an error vector value is used. As the LQR utilizes a state-space model, the controller works with all of the states instead of working only with the output.

This in reality means, that it is possible to set reference not only to the output but to all of the states—and thus control the system in its entirety.

However, there is an obvious drawback, that all of the system states in the current iteration have to be known in order to calculate the state error. As it was discussed in Chap. 4.3, even if the states are not being measured directly, they can be estimated with sufficient accuracy—the functions `getKalmanEstimate()` and `estimateKalmanState()` utilizing the Kalman filter algorithm were created for this very purpose.

It has to be kept in mind, that while the LQR offers a more advanced form of control, where it is possible to influence all of its individual states, it can be (in cases where not all of the states are being measured) dependent on the use of relatively complex algorithms such as Kalman filter.

Its ability to control the states is hidden within the LQ gain matrix $\mathbf{K}$—that is acquired through the solution of the Riccati equation. A thorough explanation of the theory behind the Riccati equation is beyond the scope of this work, however for the sake completeness a simplified explanation of the process of computing the gain matrix $\mathbf{K}$ will be provided.

The basic premise of the gain matrix $\mathbf{K}$ is, that it is being calculated in such way, that the control law defined by Eq. (5.3) in the case of zero reference (so that it becomes $\mathbf{u}_{(k)} = -\mathbf{K}\mathbf{x}_{(k)}$) minimizes the quadratic cost function defined as

$$J_{(k)} = \sum_{k=1}^{\infty} \left(\mathbf{x}_{(k)}^T \mathbf{Q} \mathbf{x}_{(k)} + \mathbf{u}_{(k)}^T \mathbf{R} \mathbf{u}_{(k)} \right), \tag{5.4}$$

where:

- $J_{(k)}$ is the quadratic cost;
- $\mathbf{Q}$ is the state penalty matrix;
- $\mathbf{R}$ is the input penalty matrix.

Here, the cost function, sometimes also called loss function, is an artificially created numerical indicator that intuitively represents the quality of control—where lower value means "better" control and higher value means "worse" control. Thus, by finding the minimum of this function in respect to the system input, such system input will be found, that will guarantee optimal control (in the sense of minimizing (5.4)). In this case it is also

called a quadratic cost function, as the state vector and input vector are squared—in a vector-compatible way, while the reason for such a form is significantly easier mathematical minimisation.

Through the penalty matrices $\mathbf{Q}$ and $\mathbf{R}$, one may specify how significantly will the values of individual states or individual inputs add to the value of the cost, and thus the priorities of the controller can be influenced. For example, if one of the values in the matrix $\mathbf{R}$ is higher than the rest, the corresponding value of calculated system input has to be smaller as to keep the overall cost low, and if one of the values in the matrix $\mathbf{Q}$ is higher than the rest, the controller will try to control the corresponding state to the reference value with equivalently higher urgency as to keep the overall cost low.

As it was mentioned previously, the continuous state-space models must be discretized in order to become implementable on digital computers. The discrete state-space representation is defined by

$$\mathbf{x}_{(k+1)} = \mathbf{A}\mathbf{x}_{(k)} + \mathbf{B}\mathbf{u}_{(k)}, \tag{5.5a}$$

$$\mathbf{y}_{(k)} = \mathbf{C}\mathbf{x}_{(k)} + \mathbf{D}\mathbf{u}_{(k)}, \tag{5.5b}$$

where:

- $\mathbf{y}_{(k)}$ is the system output vector at discrete time k;
- $\mathbf{A}$ is the system matrix;
- $\mathbf{B}$ is the input matrix;
- $\mathbf{C}$ is the output matrix;
- $\mathbf{D}$ is the feedthrough matrix.

However, most often, the $\mathbf{D}$ matrix is omitted, as there is rarely a direct correlation between input and measurement at instant k, while the second equation (5.5b) serves only to specify through matrix $\mathbf{C}$ how the individual system states are converted into the actual measurement.

The matrices $\mathbf{A}$ and $\mathbf{B}$ can be clearly seen in the linear state-space model of the FloatShield system shown in Eq. (4.16) (although in their continuous form), and they are crucial for solving the algebraic Riccati equation

$$\mathbf{P}_{(k)} = \left(\mathbf{A} + \mathbf{B}\mathbf{K}_{(k)}\right)^{\mathrm{T}} \mathbf{P}_{(k)} \left(\mathbf{A} + \mathbf{B}\mathbf{K}_{(k)}\right) + \mathbf{K}_{(k)}^{\mathrm{T}}\mathbf{R}\mathbf{K}_{(k)} + \mathbf{Q}, \tag{5.6a}$$

$$\mathbf{K}_{(k)} = \left(\mathbf{R} + \mathbf{B}^{\mathrm{T}}\mathbf{P}_{(k)}\mathbf{B}\right)^{-1} \mathbf{B}^{\mathrm{T}}\mathbf{P}_{(k)}\mathbf{A}, \tag{5.6b}$$

where:

- $\mathbf{P}_{(k)}$ is terminal state penalty matrix at discrete time k.

Note, that in this form, both Eq. (5.6a) and Eq. (5.6b) are interdependent and have to be solved at the same time.

By solving the set of these two equations, either iteratively or using one of the many advanced algorithms that can be found in MATLAB, both matrices—the LQ gain matrix

$\mathbf{K}$ and at the same time the terminal state penalty matrix $\mathbf{P}$, which represents a stable solution of algebraic Riccati equation—are obtained.

The LQ gain matrix $\mathbf{K}$ can then be used as a part of a LQ control law (5.3), while the terminal state penalty matrix $\mathbf{P}$ is a required input parameter in MATLAB function `calculateCostFunctionMPC()`, that will be more closely explained later on.

While for the needs of Arduino IDE a `getGainLQ()` function has been created that solves the equations (5.6) iteratively and outputs the $\mathbf{K}$ gain matrix, a native function `dlqr()` can be used within MATLAB that solves these equations using an advanced algorithm, and optionally outputs both $\mathbf{K}$ and $\mathbf{P}$ matrices.

5.3.1 An Integrator State

Despite being better in every aspect—assuming existence of a sufficiently accurate state-space model of the system—the LQR is not by default equipped by a means of monitoring past error as it was in the PID controller.

This fact can cause unexpected problems if—for any reason—the LQR is not able to calculate high enough or low enough input, that would achieve match between the actual states and reference states—this can manifest as a steady-state error that the controller is not able to eliminate. Additionally, a means of monitoring the error from the past, and inclusion of such error into the system input calculation are necessary, for us to be able to observe the trajectory of the input.

In order to combat this problem, a possible solution for the state-space model is to artificially add another state, that will represent the integrator within the model. If the integrator is one of the states, it will be included in the calculation of control action—so that the controller can take into account the error from the previous iterations as well.

In the case of the three state FloatShield system, the model will be extended by a fourth integrator state, which will be defined as

$$\mathbf{x^I}_{(k+1)} = \mathbf{x^I}_{(k)} + \left(\mathbf{r}_{(k)} - \mathbf{y}_{(k)}\right), \tag{5.7}$$

and after substituting the output vector with Eq. (5.5b) it becomes

$$\mathbf{x^I}_{(k+1)} = \mathbf{x^I}_{(k)} + \left(\mathbf{r}_{(k)} - \mathbf{C}\mathbf{x}_{(k)}\right). \tag{5.8}$$

This addition of the actual error into the fourth integrator state shown in (5.8) will have to be done manually—outside of the controller—in every iteration, that is, however, a fairly low price for a solution of many future problems, including steady-state errors.

The original state vector will be augmented by a fourth state simply by appending it in a following manner

$$\begin{bmatrix} \mathbf{x}_{(k+1)} \\ \mathbf{x^I}_{(k+1)} \end{bmatrix} = \begin{bmatrix} \mathbf{A} & \mathbf{0} \\ -\mathbf{C} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{x}_{(k)} \\ \mathbf{x^I}_{(k)} \end{bmatrix} + \begin{bmatrix} \mathbf{B} \\ \mathbf{0} \end{bmatrix} \mathbf{u}_{(k)} + \begin{bmatrix} \mathbf{0} \\ \mathbf{I} \end{bmatrix} \mathbf{r}_{(k)}, \tag{5.9}$$

so that the model augmented by the integrator can be referred to as

$$\widetilde{\mathbf{x}}_{(k+1)} = \widetilde{\mathbf{A}}\widetilde{\mathbf{x}}_{(k)} + \widetilde{\mathbf{B}}\mathbf{u}_{(k)} + \mathbf{r}_{(k)}, \tag{5.10}$$

where:

- $\widetilde{\mathbf{x}}_{(k)}$ is the state vector extended by the integrator at discrete time k;
- $\widetilde{\mathbf{A}}$ is the system matrix extended by the integrator;
- $\widetilde{\mathbf{B}}$ is the input matrix extended by the integrator;
- $\mathbf{r}_{(k)}$ is the reference vector at discrete time k.

When calculating the LQ gain $\mathbf{K}$, a gain that takes into consideration the fourth integrator state is obtained by using these extended matrices and vectors, and it can be denoted as extended gain with $\widetilde{\mathbf{K}}$.

The impact of the fourth integrator state on controller output can be increased or reduced—as with all other states—through the values of corresponding elements of the extended state penalisation matrix $\widetilde{\mathbf{Q}}$.

5.3.2 LQ Example in Arduino IDE

The LQ control example implementing the LQ control law represented by Eq. (5.3) has been created within the Arduino IDE in the sketch named `FloatShield_LQ` (.ino suffix). Its C++ code is provided in appendix A.8. This example follows the structure described in Chap. 5.1, with the following additions:

The **Initialisation** phase is extended by the definition of the reference array `R[]` and the section length `T` as

```
// Reference trajectory in mm
float R[] = {210,160,110,145,195,245,180,130,65,95};
int T = 1200;                              // Section length
```

followed by the definition of matrices for the LQ controller and Kalman filter—including state-space matrices, noise covariance matrices and extended gain matrix acquired through the `printBLAMatrix()` function from MATLAB.

```
// State-space model matrices A,B,C
BLA::Matrix<3,3> A = {1,0.024,0,0,0.950,0.047,0,0,0.899};
BLA::Matrix<3,1> B = {0,0.013,0.517};
BLA::Matrix<1,3> C = {1,0,0};
// Process and measurement noise covariance matrices Q_K,R_K
BLA::Matrix<3,3> Q_K = {5,0,0,0,1000,0,0,0,1000};
BLA::Matrix<1,1> R_K = {25};
// Extended LQ gain with integrator
BLA::Matrix<1, 4> K = {0.458,0.199,0.639,-0.003};
```

Then, the **Control algorithm** consists of

```
Xr(0) = R[i];                              // Select reference
y = FloatShield.sensorReadAltitude();      // Read sensor
u = -(K*(X-Xr))(0);                        // Calculate LQ
FloatShield.actuatorWrite(u);              // Actuate
//Estimate state vector X
FloatShield.getKalmanEstimate(X,u,y,A,B,C,Q_K,R_K);
X(3) = X(3)+(Xr(0)-X(0));     // Add error to integrator state
```

where initially a reference of the first state `Xr(0)` is chosen from the array `R[]` based on the elapsed number of samples, and the measurement of the ball position (altitude) in millimetres is taken and stored in `y`. Subsequently, the system input `u` is calculated through the extended LQ gain matrix `K`, based on the difference of `(X-Xr)` representing the state error. Afterwards, the input `u` provided to the device fan using corresponding method. Next, the first three system states are estimated with the `getKalmanEstimate()` FloatShield method and stored in `X`, based on the known current system input `u` and the system output `y`. Finally, the fourth state is incremented by the value of the current first state error.

Note, that the calculation of LQR output and state estimation should be ideally executed together—as to provide the actual system input `u` to the `getKalmanEstimate()` function; and the actual estimated states `X` to the LQ control algorithm at the same time. This is, however, not possible with the current separate implementation of these methods, and therefore the LQ control law has to calculate the system input from the states estimated in the previous iteration.

Figure 5.5 shows the results acquired by the `FloatShield_LQ` (.ino suffix) Arduino IDE example.

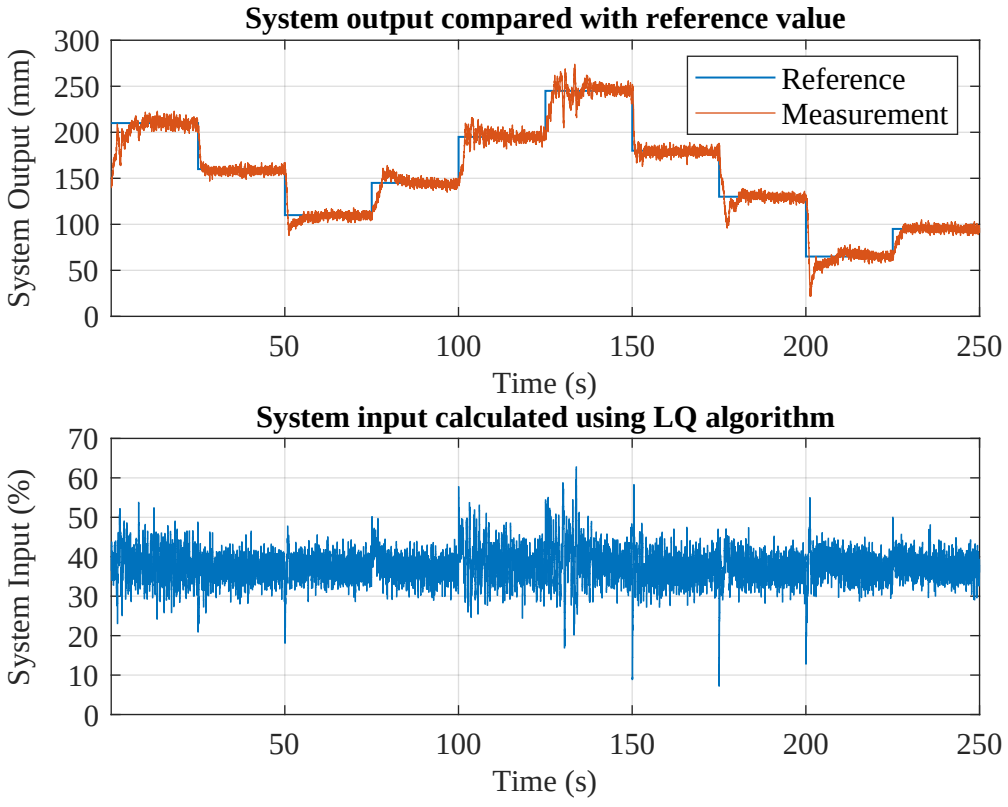


Figure 5.5: Results of the FloatShield LQ Arduino IDE example.

There are a few apparent differences between these results and those previously obtained in PID experiments. For example, the input is more noisy to the point that its reactions to the sudden reference changes are almost unrecognizable. The reference tracking by the system input is almost perfect, showing only minimal amount of deviations at higher ball positions and shortly after sudden reference changes.

Another interesting difference is, that the LQR output oscillates around the value of approximately 40%, what is significantly lower than the 55% or 53% from the PID examples. This can be explained by the fact, that the calculated system input oscillates more violently—periodically achieving values from the range of 30% up to 45% of fan power, while in the case of the PID examples the inputs were most of the time within range from 52% to 56%.

One last thing to notice is, that the system output is in the form of millimetres as opposed to percentages in this case. In the case of PID, it does not matter in what form the system input and output are, as the PID controller can manage both forms provided by FloatShield methods (assuming that the constants will change proportionally to the percentage-millimetre mapping). However in the case of LQR, the state-space model it utilizes was identified using measurements in millimetres, meaning that the LQR based on such model would not work properly if the experiment utilised measurements in percentages.

When it comes to experiment duration, the time spent on each reference level has been halved in the case of LQR, as the faster reaction and stabilisation time of the current controller tuning allowed such a reduction.

5.3.3 LQ Example in MATLAB

Within the MATLAB software environment the FloatShield LQ control example has been implemented through a script under the name `FloatShield_LQ` (.m suffix). The MATLAB code of this example is provided in appendix B.12. It abides the format presented in Chap. 5.1, with the following modifications:

The **Preparation** phase is extended by clearing the state estimating function

```
clear estimateKalmanState;      % Clears persistent variables
```

because it utilizes so-called persistent variables, which—as their name suggests—persist over function calls and therefore over iterations and even script launches. To make sure that these variables are empty at the start of the experiment, the function has to be explicitly cleared.

Reference vector `Ref` and the section length `T` are defined in the **Initialisation** phase as

```
Ref = [210,160,110,145,195,245,180,130,65,95];
T = 1200;                       % Section length
```

followed by loading the matrices for the LQR and Kalman filter—including state-space model matrices and noise covariance matrices from the external file, where they were stored after being acquired in `FloatShield_Ident_Greybox` MATLAB identification example. The system matrices `A` and `B` are then extended by a fourth integrator state by an approach proposed in Chap. 5.3.1.

```
load FloatShield_LinearSS_Discrete_Matrices_25ms
% Extend matrices with integrator state
matAhat = [matA,zeros(3,1);-matC,1];
matBhat = [matB;0];
```

Then, after defining the values of penalty matrices, the extended LQR gain matrix calculated using MATLAB native `dlqr()` function.

```
Q = diag([1,1,1e7,1e2]);          % State penalisation
R = 1e7;                          % Input penalisation
K = dlqr(matAhat,matBhat,Q,R); % Extended LQ gain
```

After the short takeoff routine, the main **Control algorithm** is defined as

```
Xr(1) = Ref(i);                                % Select reference
y = FloatShield.sensorReadAltitude(); % Read sensor
u = -K*(X-Xr);                                 % Calculate LQ
FloatShield.actuatorWrite(u);                  % Actuate
X(1:3) = estimateKalmanState(u,y,matA,matB,matC,Q_K,R_K);
X(4) = X(4)+(Xr(1)-X(1));       % Add error to integrator state
```

where initially a reference of the first state `Xr(1)` is chosen from the vector `Ref` based on the number of samples that have already passed. Then, the measurement of the ball position (altitude) in millimetres `y` is taken, and the system input `u` is calculated based on the difference of `(X-Xr)` representing the state error through the extended LQ gain matrix `K`. Subsequently the input `u` is fed to the corresponding FloatShield method that provides this value to the device fan. Afterwards, the first three system states are estimated using the `estimateKalmanState()` FloatShield method based on the known system input `u` and system output `y`, and are stored in `X`. Finally, the fourth state is incremented by the value of the current first state error.

The results of the `FloatShield_LQ` (.m suffix) example, are illustrated in Fig. 5.6.

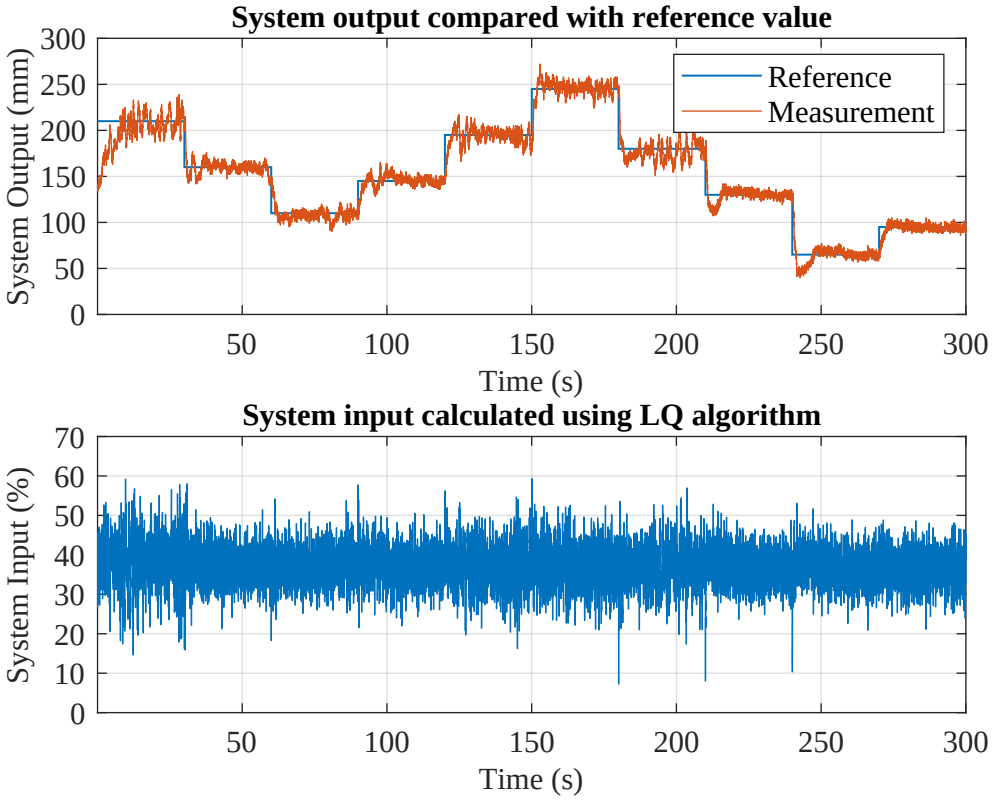


Figure 5.6: Results of the FloatShield LQ MATLAB example.

These results are almost identical to the ones acquired by the Arduino IDE LQ example, but are slightly more noisy when it comes to calculated system input—what can be attributed to the communication delay between the MATLAB and the Arduino board. Reference tracking is highly accurate, although with a small amount of oscillation present at higher parts of the tube. The example used the same parameters as its Arduino IDE counterpart and because of that, this experiment lasted approximately 5 minutes as well.

5.3.4 LQ Example in Simulink

An implementation of the LQ control algorithm represented by Eq. (5.3) is realised within the Simulink environment in a form of model named `FloatShield_LQ` (.slx suffix).

The Simulink FloatShield LQ example is described by a Fig. 5.7

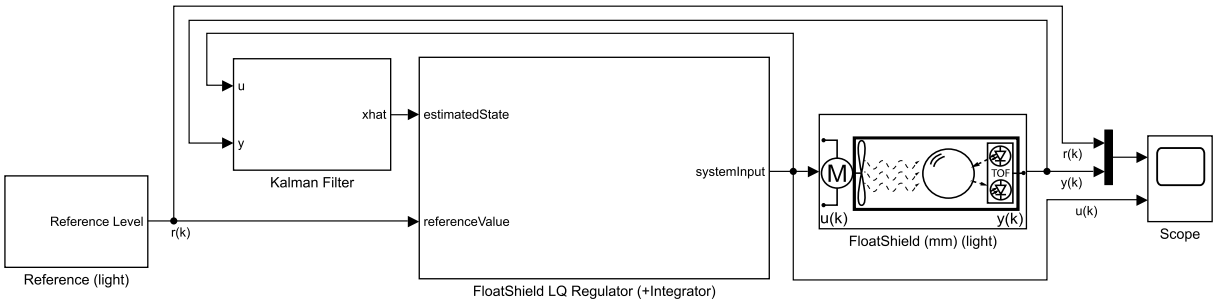


Figure 5.7: Simulink model representing FloatShield LQ example.

where the individual blocks were introduced and explained in Chap. 3.5.

Let us now review the flow of information in the Simulink FloatShield LQ example. First, the current reference level from the `Reference (light)` block is sent to the `FloatShield LQ Regulator (+Integrator)` block together with a current state vector estimate from `Kalman Filter` Simulink native block.

Consequently, the current values of reference and state estimate are processed in the `FloatShield LQ Regulator (+Integrator)` block, which manages the incrementation of the fourth integrator state and, at the same time, calculates the LQR output. This output is then sent back to the `Kalman Filter` block and also to the FloatShield system represented by the `FloatShield (mm) (light)` block, where the input is applied to the system actuator (fan).

Finally, the measurement of the ball position in millimetres is led from the system back to the `Kalman Filter` block, that based on this measurement, and also based on the system input calculated by the controller, estimates the unknown (unmeasured) system states.

At the end of example, the results are stored into the `SimulinkLQResponse` MATLAB file.

Note, that the process of calibration is realised at the beginning of the experiment through the `FloatShield (mm) (light)` block.

A plot of the results obtained by the `FloatShield_LQ` (.slx suffix) Simulink example is shown in Fig. 5.8 on the following page.

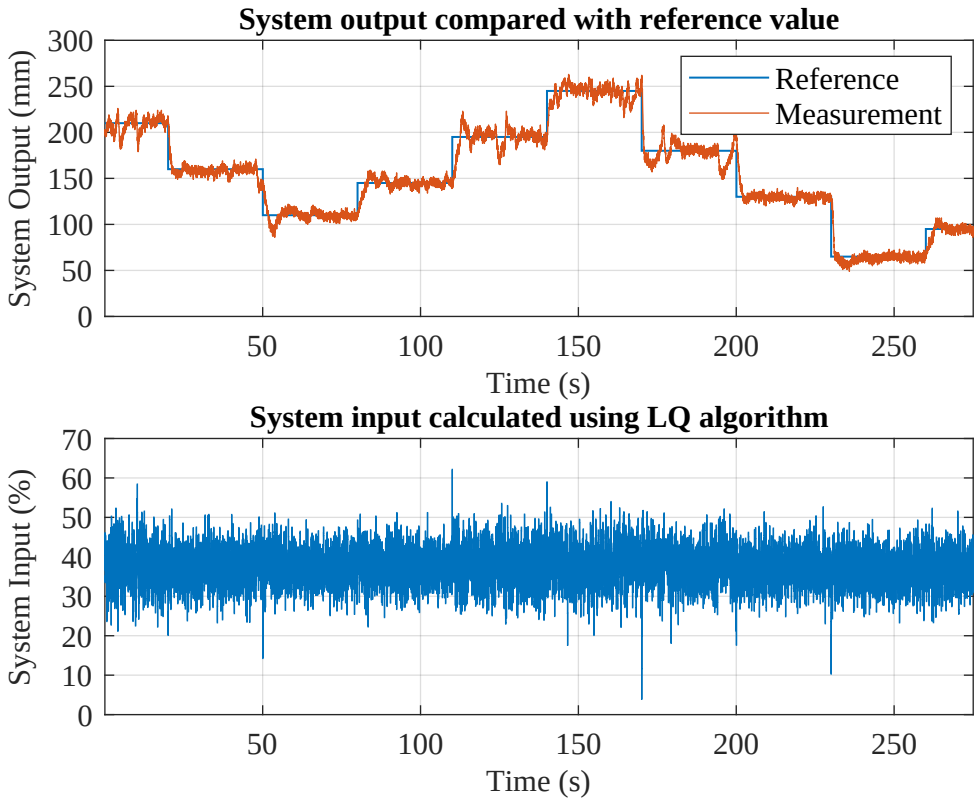


Figure 5.8: Results of the FloatShield LQ Simulink example.

The results shown in Fig. 5.8 are very similar to their Arduino IDE and MATLAB counterparts. Reference levels are being tracked accurately and there is present a similarly high amount of noise as before, while in this case it is slightly lower than in the case of MATLAB. The experiment took approximately 5 minutes from the start to the end.

5.4 MPC

Model predictive control (MPC), is the most advanced control algorithm among the three algorithms currently implemented on the FloatShield system. It proposes an ingenious idea of controlling the system in the present, based on the predicted future. Thus guaranteeing that the input in the current step will be such, as to produce the optimal outcome in the future.

Contrary to the PID controller, MPC is not restricted to be used only on SISO systems, and additionally in comparison to both PID controller and LQR, it provides a solution to the issue of the input not being within the range of process boundaries, by introducing the concept of constraints.

These constraints are defined mathematically in a form of inequations, and can be used not only to limit the values of inputs, but also of system states—giving us absolute control over the system.

The main disadvantage of MPC—assuming the existence of a sufficiently accurate state-space representation of the system—is, that it is based on a complex theory that utilizes a great number of demanding matrix operations. It makes this form of control

often difficult to implement, due to the potentially enormous (based on the level of prediction that should be achieved and on the complexity of the model) requirements on the computation hardware. In some cases, specialized algorithmic solvers are needed to enable efficient and fast real-time calculations.

A thorough explanation of the theory behind MPC is far beyond the scope of this work—as there are entire books dedicated to such efforts [20]—however, a simplified and understandable explanation of MPC should be attempted.

The problem that is being solved in MPC can be described by the equation

$$\overrightarrow{\mathbf{u}}^{*}_{(k)} = \underset{\overrightarrow{\mathbf{u}}_{(k)}}{\operatorname{argmin}} \left(\frac{1}{2} \overrightarrow{\mathbf{u}}^{T}_{(k)} \mathbf{H} \overrightarrow{\mathbf{u}}_{(k)} + \mathbf{x}^{T}_{(k)} \mathbf{G}^{T} \overrightarrow{\mathbf{u}}_{(k)} \right), \tag{5.11}$$

where:

- $\overrightarrow{\mathbf{u}}^{*}_{(k)}$ is the optimal sequence of system inputs at discrete time k;
- $\overrightarrow{\mathbf{u}}_{(k)}$ is the sequence of system inputs at discrete time k;
- $\mathbf{x}_{(k)}$ is the state vector at discrete time k;
- $\mathbf{H}$ is the Hessian matrix;
- $\mathbf{G}$ is the matrix of MPC cost function, while ${\mathbf{x}_{(k)}}^{\mathrm{T}}\mathbf{G}^{\mathrm{T}}$ is the gradient.

Based on Eq. (5.11) it can be said, that the main goal of MPC is to find such a sequence of system inputs, which will minimise the MPC cost function defined on the right side within the brackets. In other words, the minimum of the MPC cost function in respect to the vector of optimal system inputs has to be found at each sample.

The length of the sequence of system inputs depends on the value of parameter n_p known as the prediction horizon, which defines how far into the future the cost function should be minimized; i.e. how many future samples should be taken into consideration when looking for the optimal way of reaching the reference value. In Eq. (5.11) the prediction horizon is contained within the cost function matrices $\mathbf{H}$ and $\mathbf{G}$ as a part of their dimensions.

Before looking for ways of solving Eq. (5.11), first the values of matrices $\mathbf{H}$ and $\mathbf{G}$ have to be found, which will characterize the MPC cost function defined by

$$J_{(k)} = \overrightarrow{\mathbf{u}}^{T}_{(k)} \mathbf{H} \overrightarrow{\mathbf{u}}_{(k)} + 2\mathbf{x}^{T}_{(k)} \mathbf{G}^{T} \overrightarrow{\mathbf{u}}_{(k)} + \mathbf{x}^{T}_{(k)} \mathbf{F} \mathbf{x}_{(k)}, \tag{5.12}$$

where:

- $\mathbf{F}$ is the matrix of MPC cost function, which has no effect on the optimised variable.

Notice, that the cost function in MPC defined by Eq. (5.12) has a similar quadratic form as the one defined in LQR by Eq. (5.4). The quadratic form means, that the state vector and input vector are squared—in a vector-compatible way. In this case the most important fact is, that the optimisation variable $\overrightarrow{\mathbf{u}}_{(k)}$ is squared, since it is significantly easier to find the minimum of a convex parabola, than of an irregular non-convex function.

For computing the MPC cost function matrices $\mathbf{H}$, $\mathbf{G}$ and $\mathbf{F}$ a MATLAB function `calculateCostFunctionMPC()` was created. These matrices can be acquired using the recursive iterative method, and are based on the knowledge of state-space matrices $\mathbf{A}$ and $\mathbf{B}$, prediction horizon n_p and penalisation matrices of states $\mathbf{Q}$, of inputs $\mathbf{R}$ and of the final state $\mathbf{P}$. The complete theory behind them can be found in [20].

Note, that the standalone cost function defined by (5.12) and the one present in the Eq. (5.11) are the same function. However the Eq. (5.11) is a simplified form of Eq. (5.12), where only simplifications not impacting the optimal sequence of inputs were performed.

The cost function matrices $\mathbf{H}$, $\mathbf{G}$ and $\mathbf{F}$ do not change in time, and thanks to that, they have to be calculated only once at the beginning, and then only provided to the solver (assuming that the system is time-invariant).

Minimisation of the quadratic cost function is a fairly regular occurrence when it comes to optimisation of any kind, and because of that, there are many types of algorithmic solvers that specialize in minimizing the equations defined in a quadratic form, such as ours in Eq. (5.11).

Two different ways of solving the MPC problem will be utilized here—a native solver under the name `quadprog()` will be used in the MATLAB environment, and in the Arduino IDE the so-called μAO-MPC package will be used, that can be described as a free code generation software for linear model predictive control [22].

The inner workings of many solvers is oftentimes a treasured secret, as the problem of optimization is still very actual, but thankfully in the case of MATLAB, `quadprog()` can be used as a part of the MathWorks licence, even if the details of its functioning are not entirely disclosed. When it comes to μAO-MPC, it provides the user with the ability to generate C code files through a Python (programming language) script, that include the iterative MPC algorithm. Despite the C code not being perfectly optimised to be applied in embedded systems with limited memory, it enables us to attempt the MPC control also on the Arduino as well.

Generally, the inputs to the solver of the quadratic cost function Eq. (5.11), are the matrix $\mathbf{H}$ and the result of matrix multiplication $\mathbf{x}_{(k)}^T\mathbf{G}^T$, while the task of the solver is then to look for the optimal sequence of inputs $\overrightarrow{\mathbf{u}}_{(k)}$.

Additionally, as optional, albeit very important arguments, the matrices $\mathbf{A}_c$, $\mathbf{b}_0$ and $\mathbf{B}_0$ can be defined, giving the input and state constraints in a form of an inequation

$$\mathbf{A}_c\overrightarrow{\mathbf{u}}_{(k)} \leq \mathbf{b}_c, \tag{5.13}$$

where

$$\mathbf{b}_c = \mathbf{b}_0 + \mathbf{B}_0\mathbf{x}_{(k)}. \tag{5.14}$$

The inequation (5.13) represents a mathematical condition for the solver, that has to be met throughout the process of minimisation. Through it, either the system input constraints (utilising matrices $\mathbf{A}_c$ and $\mathbf{b}_0$) or system state constraints (utilising matrices $\mathbf{A}_c$, $\mathbf{b}_0$ and $\mathbf{B}_0$) can be defined, with an option to set both types of constraints at the same time. Matrix $\mathbf{b}_c$ is defined in (5.14), only as an convenient way of representing the right side of the constraint-defining inequation (5.13).

Computation of these three constraint-defining matrices is also realised through the recursive iterative method, and is described in detail in [20]. Based on this approach,

these matrices can be obtained through the use of `applyConstraintsMPC()` MATLAB function, that expects to get the values of lower and upper limits of the constrained system inputs and/or states, the length of the prediction horizon n_p, and the state-space matrices $\mathbf{A}$, $\mathbf{B}$.

In case only the system input is being constrained, these matrices can be calculated only once at the beginning of the experiment, while in the case of state or combined constraints the $\mathbf{B}_0\mathbf{x}_{(k)}$ multiplication has to occur at every sample due to the changes in the state vector.

The longer the prediction horizon n_p is, the more steps the problem (5.11) is solved into the future, thus the longer sequence of inputs $\overrightarrow{\mathbf{u}}^*_{(k)}$ will be found. This guarantees optimal control over a longer period of time. However, the matrix operations with larger matrices will slow the solving process down, what can be in some cases highly undesirable—a reasonable value of the prediction horizon has to be chosen, as to get satisfactory results while keeping the solving process fast.

By solving the problem defined by Eq. (5.11), while abiding (5.14), a sequence of optimal future system inputs $\overrightarrow{\mathbf{u}}^*_{(k)}$ over the horizon of n_p steps into the future is calculated. What seems not so logical is, that only the first member of the sequence is actually applied to the actuator at the current discrete time k. In the next iteration, the entire solving process will start again, and as a result, a new optimal sequence of inputs $\overrightarrow{\mathbf{u}}^*_{(k+1)}$ will be found again.

This "moving" horizont guarantees, that the system input in the next n_p steps will not compromise the quality of control, and as this holds true in every iteration, the system should theoretically retain optimal control infinitely—as it will always calculate with the future in mind.

To summarise, the solver—in this case either `quadprog()` or μAO-MPC—has to evaluate the problem defined by Eq. (5.11), while meeting the condition expressed in (5.13). The result will be a sequence of optimal inputs $\overrightarrow{\mathbf{u}}^*_{(k)}$, from which only the first member is applied to the system. This process occurs in every sample, and to make sure that the total execution time is not too high, a suitable prediction horizon n_p has to be used. The matrices utilized in (5.11) and (5.13) can be obtained through the `calculateCostFunctionMPC()` and `applyConstraintsMPC()` MATLAB functions, which are described in [20] in detail.

5.4.1 MPC Example in Arduino IDE

The MPC algorithm defined by Eq. (5.11) and Eq. (5.13) is implemented within the Arduino IDE in the sketch named `FloatShield_MPC` (.ino suffix). Its code can be found in appendix A.9.

The μAO-MPC algorithm proposed in [22] is being used as a quadratic solver. As opposed to `quadprog()`, μAO-MPC generates the entire MPC problem, along with the solver itself. From a user perspective it is realised through a Python script, where the parameters of the system, and also, certain information for the solver have to be provided, such as sampling period T_s, state-space matrices $\mathbf{A}$, $\mathbf{B}$, lower and upper limits of the system inputs and/or the system states, and finally state penalty matrix $\mathbf{Q}$, input penalty matrix $\mathbf{R}$, and terminal state penalty matrix $\mathbf{P}$ are given.

In the case of FloatShield, only the input constraints will be applied—for the calculated

system input to fit into the percentual range of 0%–100%.

For the FloatShield MPC experiment, a different Arduino board than in the previous experiments had to be used, as the Arduino UNO did not have a sufficient amount of flash memory for storing the μAO-MPC generated C code. The highest prediction horizon that could be used for the Arduino UNO board was $n_p = 2$, and while that technically still is MPC, it does not truly take the full advantage of the MPC algorithm. For this reason, the MPC FloatShield experiment was realised on the Arduino MEGA2560 board, where the highest prediction horizon (implemented within the Python-generated C code) that could fit into the flash memory of the board was $n_p = 15$, although with 95% of dynamic memory taken up by the sketch and a warning about potential stability problems. For those reasons, it was deemed best to use a prediction horizon $n_p = 10$, that took only 57% of Arduino MEGA dynamic memory, leaving enough room for other calculations. The example respects the format proposed in Chap. 5.1, with the following differences:

The μAO-MPC Python-generated C code is included in the **Preparation** phase through the preprocessor directive

```
// Include muAO-MPC Python compiled C files
#include "FloatShield_muAO-MPC/FloatShield_MPC.h"
```

where the μAO-MPC files contained in `FloatShield_MPC.h`, represent the solving algorithm custom-generated for the FloatShield system—through the matrices provided to the Python script—that will be used to find an optimal sequence of inputs.

The **Initialisation** phase is extended by the definition of reference array `Ref[]` and the section length `T` as

```
// Reference trajectory in mm
float Ref[] = {210,160,110,145,195,245,180,130,65,95};
int T = 2400;                              // Section length
```

followed by the definition of Kalman filter matrices—including state-space model matrices and noise covariance matrices acquired through the `printBLAMatrix()` function from MATLAB. Additionally, an object `ctl` representing the MPC solver is created, through which the calculated input sequence may be obtained, and the settings of the solver can be accessed.

```
// State-space model matrices A,B,C
BLA::Matrix<3,3> A = {1,0.024,0,0,0.950,0.047,0,0,0.899};
BLA::Matrix<3,1> B = {0,0.013,0.517};
BLA::Matrix<1,3> C = {1,0,0};
// Process and measurement noise covariance matrices Q_K,R_K
BLA::Matrix<3,3> Q_K = {5,0,0,0,1000,0,0,0,1000};
BLA::Matrix<1,1> R_K = {25};
extern struct mpc_ctl ctl;     // Create MPC solver object
```

The main **Control algorithm** (after the short takeoff routine) is defined

```
Xr(0) = Ref[i];                            // Select reference
y = FloatShield.sensorReadAltitude();      // Read sensor
mpc_ctl_solve_problem(&ctl,X);             // Calculate MPC
u = ctl.u_opt[0];                          // Store first member
FloatShield.actuatorWrite(u);              // Actuate
//Estimate state vector X
```

```
FloatShield.getKalmanEstimate(X,u,y,A,B,C,Q_K,R_K);
X(3) = X(3)+(Xr(0)-X(0));       // Increment summation state
```

where at the start a reference of the first state `Xr(0)` is chosen from the array `Ref[]` based on the number of samples that have elapsed, and the measurement of the ball position (altitude) in millimetres `y` is taken. Then, the optimal sequence of inputs `u_opt` is calculated through the included MPC solver, based on the value of state vector `X` from the previous iteration. Only the first element of the sequence is stored into `u`. Afterwards, the input is fed to the FloatShield fan through the respective method. Subsequently, the first three system states are estimated and stored in `X`, based on the known input `u` and system output `y` through the `getKalmanEstimate()` FloatShield method. Finally the fourth state is incremented by the first state error.

A plot of the results acquired by the `FloatShield_MPC` (.ino suffix) example is shown in Fig. 5.9.

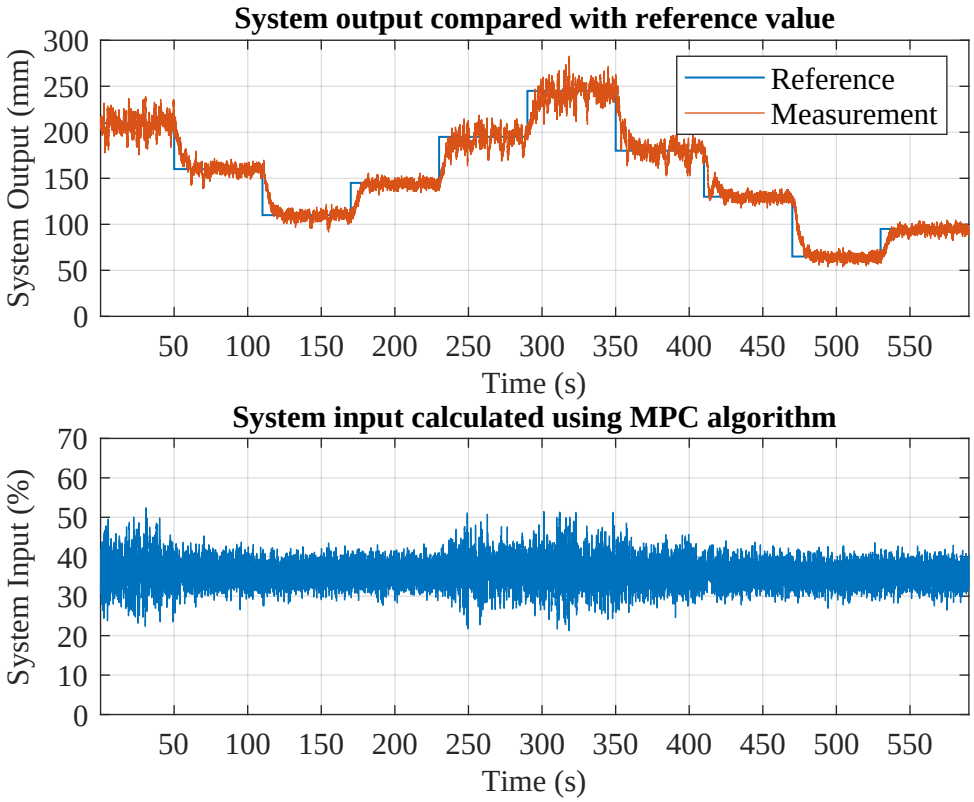


Figure 5.9: Results of the FloatShield MPC Arduino IDE example.

In the case of MPC, while the reference tracking is of high accuracy and there are no unwanted overshoots, the calculated system input is significantly less noisy than in the case of the LQ examples. The only thing that is worse than in LQ examples, is the reaction speed—the MPC algorithm had to be used with a section length of 2400 samples, as it was used in PID examples, since its reactions were slower and it required longer time to reach steady state. This behaviour can be possibly mitigated by a better tuning. The experiment took approximately 10 minutes, similarly to the PID. In this case, the system input is being held at even lower value than in the case of LQ—an approximately constant 37% of power, what is probably caused by the present oscillation.

5.4.2 MPC Example in MATLAB

Within the MATLAB software environment the FloatShield MPC example has been implemented through a script named `FloatShield_MPC` (.m suffix). The MATLAB code of this example is provided in appendix B.13. It abides the format presented in Chap. 5.1, with the following modifications:

The MATLAB native `quadprog()` used in this example as a quadratic solver, and the example itself is executed on the Arduino UNO board—as in this case, only the MATLAB server is being uploaded onto the board, so there are no problems with lack of memory. The example adheres to the structure described in Chap. 5.1, with the following additions:

The **Preparation** phase is extended by clearing the state estimating function

```
clear estimateKalmanState;     % Clears persistent variables
```

to make sure that the persistent variables are empty at the start of the experiment.

The reference vector `Ref` and the section length `T` are defined in the **Initialisation** phase as

```
Ref = [210,160,110,145,195,245,180,130,65,95];
T = 2400;                      % Section length
```

followed by loading Kalman filter matrices from the external file. The system matrices `A` and `B` are then extended by a fourth integrator state through an approach proposed in Eq. (5.9).

```
load FloatShield_LinearSS_Discrete_Matrices_25ms
% Extend matrices with integrator state
matAhat = [matA,zeros(3,1);-matC,1];
matBhat = [matB;0];
```

Then, the terminal penalty matrix is calculated using the MATLAB native `dlqr()` function, based on these extended matrices.

```
Q = diag([1,1,1e7,1e2]);       % State penalisation
R = 1e7;                       % Input penalisation
% Get final state penalisation matrix P
[K,P] = dlqr(matAhat,matBhat,Q,R);
```

This is followed by the definition of MPC parameters, such as input constraints and prediction horizon, and by the calculation of cost function defining matrices and constraint inequation matrices.

```
uL = 0;          % Lower input limit
uU = 100;        % Upper input limit
Np = 10;         % Prediction horizon
[H,G,F] = calculateCostFunctionMPC(matAhat,matBhat,Np,Q,R,P);
% Set input constraints onto the system
[Ac,b0] = applyConstraintsMPC(uL,uU,Np);
H = (H+H')/2;  % Create symmetric Hessian matrix
% Specify the used solver algorithm
opt.SolverName = 'trust-region-reflective';
```

Additionally, the Hessian matrix is symmetrized and an alternative solver algorithm utilized by the `quadprog()` is specified—as the default algorithm was not able to solve the MPC problem within the sampling period.

Afterwards, the **Control algorithm** consists of

```
Xr(1) = Ref(i);                              % Select reference
y = FloatShield.sensorReadAltitude(); % Read sensor
% Solve MPC problem, find sequence of optimal inputs
u(:,1) = quadprog(H,G*X(:,1),Ac,b0,[],[],[],[],[],opt);
FloatShield.actuatorWrite(u(1));          % Actuate
X(1:3) = estimateKalmanState(u(1),y,matA,matB,matC,Q_K,R_K);
X(4) = X(4)+(Xr(1)-X(1)); % Add error to integrator state
```

where initially a reference of the first state `Xr(1)` is chosen from the vector `Ref` based on the elapsed number of samples. Then, the measurement of the ball position (altitude) in millimetres `y` is taken. Subsequently, the sequence of optimal inputs `u` is calculated through the `quadprog()` function, based on the state vector from previous iteration `X` ; the matrices representing the cost function `H`, `G`; and the matrices representing input constraints `Ac`, `b0`. Afterwards the first element of the calculated input sequence is fed to the corresponding FloatShield method providing it to the fan of the device. Additionally, the first three system states are estimated using the `getKalmanEstimate()` FloatShield method and stored in `X`, based on the current system input `u` and system output `y`. Finally, the fourth state is incremented by the value of the current first state error.

The results of `FloatShield_MPC` (.m suffix) example are illustrated in Fig. 5.10.

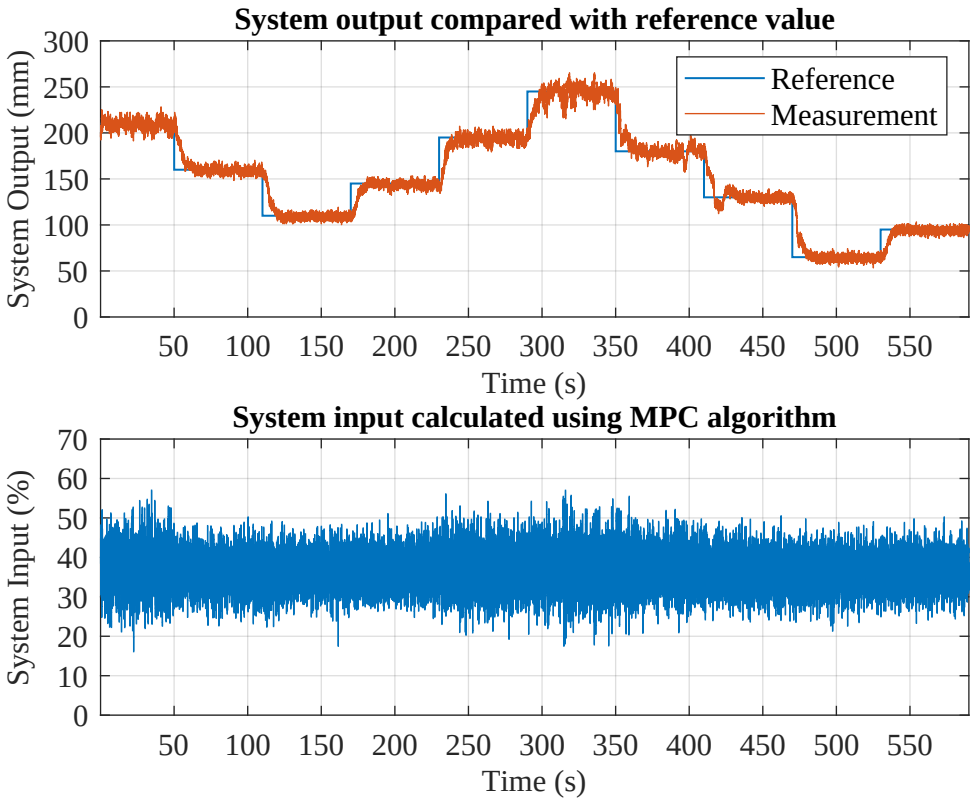


Figure 5.10: Results of the FloatShield MPC MATLAB example.

These results are almost identical to the results acquired by the Arduino IDE MPC example—the output tracks the reference very accurately and there are no overshoots, while the system input is somewhat noisy. In this case, however, the amount of noise is higher than in the case of the Arduino IDE MPC example, what can be caused—as in previous cases—due to the delayed communication between MATLAB and the Arduino board. The experiment took approximately 5 minutes in this case as well.

Note, that while it is possible to use the `quadprog()` on the Arduino board through MATLAB, it is yet not possible to do so through Simulink, as the Simulink does not allow transcription of the `quadprog()` into C code. Because of this fact, it is not yet possible to create the MPC example on the Simulink software platform.

5.5 Comparison

In this chapter, the aforementioned forms of control implemented on the FloatShield device, specifically the PID controller described in Chap. 5.2, the LQR described in Chap. 5.3 and MPC described in Chap. 5.4, will be compared in terms of practical use for embedded systems, from three different standpoints: complexity, performance, and memory management.

Note, that the following statements are based on the results obtained from the sample experiments in the previous chapters, using the specified tuning.

It will be assumed that, the required parameters and matrices for each algorithm can be calculated externally, for example in MATLAB, and then provided as a part of the code—as it was done in the previous cases with Arduino IDE matrices.

- **PID controller** from the standpoint of complexity is the simplest of the three forms of control, as it does not require any knowledge of the system and can be realised through one relatively simple equation (5.2), where the sum can be implemented as a static variable that is incremented with error in every iteration, and the differentiation can be approximated from the two adjacent measurements. The most difficult part here is finding the right combination of the three PID constants, that will provide stable and accurate control of the system. That can be done either heuristically, or by using quantitative methods, such as the Ziegler-Nichols method.

  When it comes to performance of the PID controller, it depends on the quality of the tuning heavily—on the choice of the PID constants. In our case the constants were acquired through manual tuning, and in the end, provided satisfactory results. The PID control of the FloatShield device was relatively accurate, although rather slow, but even with slower reaction time the output overshoots could not be avoided. For more complex systems, a compromise between the reaction speed and accuracy has to be made, as rarely both can be achieved through a simple PID. It can be stated, that PID is better suited for simpler, preferably linear systems.

  As the PID controller can be implemented through a single line of code, from the memory management standpoint it is perfect, as it does take up (including its sum variable) very little memory. It is therefore ideal, in cases of very limited amount of available memory for the program.

- **LQ regulator** is more complex than PID, as it does require the knowledge of the state-space model of the system. However, if the model is already known, it can be implemented through a relatively simple equation (5.3). The problem here is, that it is a matrix equation, and there have to be tools enabling at least the operations of matrix addition and multiplication present on the embedded device. However in the case of the FloatShield, and due to the fact that not all system states are being measured, a Kalman filter algorithm had to be implemented. This brought more matrix operations and consequently more complexity to the control process.

  When it comes to the performance of the LQR, it depends on the accuracy of the state-space model, and on the choice of the penalisation matrices. In our case, the penalisation matrices were chosen by trial and error. The LQ control of the FloatShield device was very accurate and also quite fast, while the output overshoots were minimal. It can be argued, that LQR is best suited for processes where the reaction speed is crucial, where a reasonable level of accuracy should be also achieved.

  From the standpoint of memory management, the LQR in and of itself would be very good, as it can be implemented through a single (albeit matrix) equation. However, as in our case, an additional and significantly lengthier algorithm was needed, such that the control algorithm takes up a moderate amount of board flash memory. It, therefore, might be suitable for MIMO systems where the coupling of the individual outputs is strong, or in the cases where all the system states are already being measured.

- **MPC** is the most complex of the three forms of control, as it requires the knowledge of the state-space model of the system, and it can be realised only through a very complex matrix equation (5.11), that requires a specialised algorithmic solver. As solving the MPC problem relies on the knowledge of the system states, in this case a Kalman filter algorithm was also necessary to be implemented, what only increased the level of complexity. In this case, it is absolutely crucial that the tools for all matrix operations will be present on the embedded system. The solver can then be realised through a included external code, in our case using the μAO-MPC solver algorithm in a form of an automatically generated C code.

  When it comes to the performance of the MPC, it depends on more factors, such as the accuracy of the state-space model, the choice of penalisation matrices, length of prediction horizon and the solver used to minimise the MPC cost function. In our case, the same penalisation matrices were used as in LQ control. The prediction horizon was chosen with the limited board memory and processing power in mind, and the μAO-MPC was picked based on good prior experience, and also because there are not that many quality and freely available solvers. The MPC control on the FloatShield device was very accurate although rather slow, while the output overshoots were practically nonexistent. It can be argued that MPC is best suited for processes where the quality of control is crucial, while reaction speed is only secondary.

  From the standpoint of memory management, MPC is absolutely terrible, as it depends on the use of an external solver that in itself, based on the length of the prediction horizon and the order of the model, can take up all the available memory

on the embedded device. Additionally, in our case, a Kalman filter algorithm had to be used, what only made the situation worse. The problems with memory then have to be mitigated by either choosing shorter prediction horizons, or limiting the internal iterations of the MPC solver, while both of these actions compromise the quality of the results of the MPC algorithm. It is, therefore, not ideal to use in embedded systems, unless the amount of available memory is adequately large. Let us note, that some alternative MPC formulations can be more suited for embedded use.

Based on the experience from previous examples, the PID controller was the least complex and most memory efficient, while providing good results, albeit with slower reaction time and frequent overshoots; the LQR was moderately complex and fairly memory efficient while providing very good results with fast reaction time and minimal overshoots; and MPC was the most complex and least memory efficient while providing excellent results with no overshoots albeit with slower reaction time that can be attributed to tuning.

Therefore, it can be generally stated, that for embedded systems with a low amount of available memory and for the control of simpler (preferably) linear systems, PID is the best choice, while for embedded systems with a moderate amount of available memory and for the control of more complex systems LQR might be the better option. For embedded systems with a high amount of available memory and for the control of very complex systems MPC will be the best choice. In the case of LQR and MPC, the state-space model of the system has to be known. As an approximate reference for memory capacity, the Arduino UNO board can be used, that was utilized in majority of the examples. It has 32 kB of flash memory for storing programs—what can be considered a moderate amount of memory. As an example of high amount of available memory the Arduino DUE board can be used, that is equipped with 512 kB of flash memory for storing programs.

Note, the reaction speed should generally not be used as a deciding factor between these forms of control, as it is heavily dependent on the used PID constants or penalisation matrices—and therefore can be potentially increased or decreased, based on the requirements.

Additionally, especially in the case of MPC, the computing power plays a significant role, and based on the required sampling period of the controlled process, it might become the limiting factor of its use.

Chapter 6

Conclusion

The presented master's thesis fully accomplished the intended goals. Various algorithms, software platforms and approaches to the problem solving are introduced by practical examples, in a user friendly way. The fairly extensive theoretical part contains the necessary knowledge base to programming and simulations. The practical demonstration of ball levitating in a vertical tube combined a wide range of expertise from various areas of mechatronics. Such an approach enables design of different examples, which can be used in the education process, making it more attractive for students. The low cost elements of the system make it affordable for the wide public.

The open-source air levitation device in form of a shield presented in this thesis, can be manufactured for a price of less than 30 €. After it is mounted to one of the popular Arduino boards, it becomes an excellent simulation and presentation tool with a strong educational potential.

Knowledge and experience concerning a wide variety of fields such as electronics, PCB design, CAD, 3D printing, universal programming languages (C++, Python), scientific programming languages (MATLAB, Simulink), system modeling and identification, Kalman filtering, forms of system control, data processing and potentially many more can be obtained by working with this device.

Despite the fact that the original FloatShield R1 design has not been improved upon by R2 in the end, it can be stated that the original hardware was employed to its full potential.

In a future R3 release—hopefully inspired by the R3 proposed in this thesis—the FloatShield device should be finally compatible with 3.3 V Arduino boards, what will allow its use on more powerful boards such as the Arduino DUE, where even more complex experiments, utilizing even more sophisticated algorithms, can be conducted.

The work on the FloatShield R1 device culminated in the form of a scientific paper, that will be published under the title ***FloatShield: An Open Source Air Levitation Device for Control Engineering Education*** at the 21st IFAC World Congress in Berlin [23].

Bibliography

[1] G. Takács, M. Gulan, J. Bavlna, R. Köplinger, M. Kováč, E. Mikuláš, S. Zarghoon, and R. Salíni. HeatShield: a low-cost didactic device for control education simulating 3D printer heater blocks. In *Proceedings of the 2019 IEEE Global Engineering Education Conference*, pages 374–383, Dubai, United Arab Emirates, April 2019.

[2] G. Takács, T. Konkoly, and M. Gulan. Optoshield: A low-cost tool for control and mechatronics education. In *Proceedings of the 12th Asian Control Conference*, pages 1001–1006, Kitakyushu-shi, Japan, Jun 2019.

[3] Gergely Takács, Jakub Mihalík, Erik Mikuláš, and Martin Gulan. MagnetoShield: Prototype of a low-cost magnetic levitation device for control education. In *Proceedings of the 2020 IEEE Global Engineering Education Conference (EDUCON)*, pages 1516–1525, Porto, Portugal, April 2020.

[4] Gergely Takács, Martin Vríčan, Erik Mikuláš, and Martin Gulan. An early hardware prototype of a miniature low-cost flexible link experiment. In *Proceedings of the 27th International Congress on Sound and Vibration*, pages 1–8, Prague, Czech Republic, July 2020.

[5] S. R. Jernigan, Y. Fahmy, and G. D. Buckner. Implementing a remote laboratory experience into a joint engineering degree program: Aerodynamic levitation of a beach ball. *IEEE Transactions on Education*, 52(2):205–213, May 2009.

[6] T. Barlas and M. Moallem. *Developing FPGA-based Embedded Controllers Using Matlab/Simulink*, chapter 27, pages 543–556. IntechOpen, Rijeka, Croatia, 2010.

[7] J. Chacon, J. Saenz, L. Torre, J. M. Diaz, and F. Esquembre. Design of a low-cost air levitation system for teaching control engineering. *Sensors*, 17(10), 2017.

[8] A. Dellah, P. Wild, and B. Surgenor. A laboratory on the microprocessor control of a floating ping pong ball. In *Proceedings of the ASEE Annual Conference & Exposition*, pages 5.32.1–5.32.8, St. Louis, Missouri, June 2000.

[9] D. M. Ovalle and L. F. Combita. Engaging control systems students with a pneumatic levitator project. In *2019 IEEE Global Engineering Education Conference*, pages 1093–1099, April 2019.

[10] M. Schaefer, D. Escobar, and H. Roth. Nonlinear identification and controller design for the air levitation system. In *2019 22nd International Conference on Control Systems and Computer Science*, pages 18–23, May 2019.

[11] J. M. Escano, M. G. Ortega, and F. R. Rubio. Position control of a pneumatic levitation system. In *2005 IEEE Conference on Emerging Technologies and Factory Automation*, volume 1, pages 523–528, Sep. 2005.

[12] D. A. Visan, I. Lita, I. B. Cioc, and R. M. Teodorescu. A new approach for aerodynamic levitation based position control using digital image processing and data acquisition. In *2015 7th International Conference on Electronics, Computers and Artificial Intelligence*, pages AE–19–AE–22, June 2015.

[13] E. Chołodowicz and P. Orłowski. Low-cost air levitation laboratory stand using MATLAB/Simulink and Arduino. *Pomiary Automatyka Robotyka*, 4(21):33–39, 2017.

[14] J. C. Kuzhandairaj. Development,control and testing of an air levitation system for educational purpose. Master's thesis, Politecnico Milano, Milan, Italy, 2018. 854576.

[15] D. Chaos, J. Chacón, E. Aranda-Escolástico, and S. Dormido. Robust switched control of an air levitation system with minimum sensing. *ISA Transactions*, 2019. In press, refer to `10.1016/j.isatra.2019.06.020`.

[16] T. Benson. Glenn research center - shape effects on drag. Online, 2014. Available at `https://wright.nasa.gov/airplane/shaped.html`, [accessed 10.5.2020].

[17] J. Saenz, J. Chacon, L. Torre, and S. Dormido. An open software - open hardware lab of the air levitation system. *IFAC-PapersOnLine*, 50(1):9168–9173, 2017. 20th IFAC World Congress.

[18] A. Vargová. *Matrix operations on programmable microcontrollers with the AVR architecture*. Bachelor's thesis, Slovak Technical University in Bratislava, Faculty of Mechanical Engineering, Bratislava, Slovak Republic, May 2019.

[19] P. Goddard. Goddard consulting - a simple Kalman filter in Simulink. Online, 2011. Available at `https://goddardconsulting.ca/simulink-kalman-filter.html`, [accessed 10.5.2020].

[20] Gergely Takács and Martin Gulan. *Základy Prediktívneho Riadenia*. Spektrum STU, Bratislava, Slovakia, 1. edition, 2018. In Slovak language. (Fundamentals of Predictive Control).

[21] V. Illingworth. *The Penguin Dictionary of Physics*. Penguin Books, London, 1991.

[22] P. Zometa, M. Kögel, and R. Findeisen. muAO-MPC: A free code generation tool for embedded real-time linear model predictive control. In *Proc. American Control Conference (ACC), 2013*, pages 5340–5345, Washington D.C., USA, 2013.

[23] Gergely Takács, Peter Chmurčiak, Martin Gulan, Erik Mikuláš, Jakub Kulhánek, Gábor Penzinger, Miloš Podbielančík, Martin Lučan, Peter Šálka, and Dávid Šroba. FloatShield: An Open Source Air Levitation Device for Control Engineering Education. In *Proceedings of the 21st IFAC World Congress*, pages 1–8, Berlin, Germany, July 2020.

Resume

Záverečná práca sa začína krátkym úvodom, v ktorom je predstavená základná idea stojaca za vznikom prístroja FloatShield a tiež mnohých ďalších zariadení projektu AutomationShield pod ktorý zariadenie patrí – vytvoriť unikátnu výučbovú pomôcku, ktorá umožní realizovať výučbový proces zážitkovou formou a tým zefektívni priebeh osvojenia si dôležitej, ale pre študentov často ťažko pochopiteľnej teórie vyučovanej na technických vysokých školách. Takéto pomôcky sú vzhľadom na neustále rastúce zastúpenie moderných technológií v bežnom živote čím ďalej tým viac potrebné, avšak stále iba veľmi ťažko dostupné – čo sa čiastočne snaží napraviť už spomenutý open-source projekt.

Prvá kapitola práce popisuje pôvod prístroja FloatShield. V jej prvej časti je vysvetlený spôsob realizácie zariadení projektu AutomationShield – prostredníctvom rozširujúcich modulov známych pod názvom "štíty". Štíty vo všeobecnosti slúžia na pridávanie rôznych nových funkcií do mikropočítačov Arduino, ako napríklad bezdrôtová komunikácia s inými zariadeniami, spracovanie údajov z rôznych senzorov, alebo riadenie činnosti pridaných akčných členov. Čo sa však týka štítov vyvíjaných v rámci tohto projektu, ide o veľmi špecializované prístroje, ktoré na rozdiel od bežne dostupných jednoúčelových štítov umožňujú vykonávanie špecifických, fyzikálno-mechanických experimentov zameraných na výučbu mechatroniky, identifikácie systémov a riadenia systémov, na príslušných vysokoškolských technických odboroch. Okrem toho, majú takéto zariadenia veľký potenciál stať sa objektom vedeckých článkov, ako napríklad tomu bolo v prípade [1–4]. Primárny typ experimentov na ktoré je dané zariadenie určené, je vo väčšine prípadov charakterizovaný prvou časťou jeho samotného názvu – napríklad prístroj FloatShield je špecializovaný na experimenty súvisiace so vznášaním sa a levitáciou. Vďaka štandardizovaným rozmerom dosiek Arduino, štíty s nimi kompatibilné musia tiež dodržiavať tieto veľkostné obmedzenia. Malá veľkosť štítov je však často spojená s relatívne nízkou výrobnou cenou, pohybujúcou sa najviac v niekoľkých desiatkach eur. Napríklad jeden z prvých štítov projektu, takzvaný OptoShield, sa dá vyrobiť za cenu nižšiu ako 3€ [2]. Okrem tvorby samotných zariadení sa projekt zaoberá tiež tvorbou im prislúchajúcich softvérových nástrojov, vo väčšine prípadov realizovaných prostredníctvom knižníc, ktoré obsahujú množstvo preddefinovaných funkcii. Zahrnutím týchto knižníc do knižnicami podporovaného vývojového prostredia, získa užívateľ zariadenia prístup k týmto funkciám, ktoré môže používať na jeho jednoduchšiu prevádzku. Skutočnosť že tento projekt sa zaraďuje do kategórie open-source znamená, že všetka dokumentácia, výrobné výkresy štítov a softvérová podpora sú voľne prístupné komukoľvek kto o ne prejaví záujem na webovej stránke projektu. Otvorenosť projektu umožňuje širšie využitie jeho zariadení a tým aj následnú potenciálnu spoluprácu na ich vývoji.

V druhej časti kapitoly je priblížený spôsob vývoja a vylepšovania týchto zariadení prostredníctvom konceptu verzií alebo vydaní – pri používaní zariadenia nieje ojedinelé si všimnúť určitý nedostatok alebo vadu, ktorú by bolo dobré odstrániť, keďže však zmena v hardvéri nieje jednoduchá, oprava sa realizuje až po tom, čo sa nazhromaždí dostatočné množstvo potenciálnych zmien a vylepšení hardvéru. Takáto opravená a vylepšená verzia je následne označená poradovým číslom jej vzniku, a predstavuje ďalšie vydanie zariadenia. Jednotlivé vydania zariadenia potom môžu byť chápané ako varianty daného zariadenia, niekedy na pohľad iba s malými rozdielmi medzi sebou, no často však významnými.

V tretej časti kapitoly je predstavený koncept prístroja FloatShield – prístroja umožňujúcemu experiment levitácie loptičky v prúde vzduchu – a taktiež sú v nej uvedené odkazy na podobné zariadenia s podobným účelom. Experiment levitácie objektu v prúde vzduchu je celosvetovo známy, pravdepodobne vďaka jeho vizuálnej a názornej povahe, no napriek tomu že pôsobí jednoducho, skrýva v sebe mnohé úskalia, ako napríklad výrazne nelineárne správanie tohto systému. Vo všeobecnosti zariadenie umožňujúce takýto typ experimentu pozostáva zo: vznášajúceho sa objektu (najčastejšie v tvare gule) vyrobeného z ľahkého materiálu; zdroja prúdu vzduchu (najčastejšie elektrického ventilátora) pomocou ktorého bude objekt nadnášaný; senzora umožňujúceho sledovanie polohy objektu (najčastejšie vhodne situovaného infračerveného senzora vzdialenosti) poskytujúceho spätnú väzbu o polohe objektu; riadiacej jednotky (realizovanej rôznou kombináciou stolného počítača a mikropočítača) kde budú senzormi získané údaje vyhodnocované a na ich základe sa určí vhodný akčný zásah zdroja prúdu vzduchu. Dizajn prístrojov sa môže líšiť, avšak každé podobné zariadenie muší obsahovať tieto spomenuté časti.

V štvrtej časti kapitoly je podrobne opísaný a znázornený dizajn pôvodného vydania zariadenia FloatShield, spolu s uvedením významu jednotlivých komponentov v súvislosti s uvedenou elektrickou schémou. Telo štítu je tvorené doskou plošných spojov (DPS) na ktorú sú jednotlivé elektrické komponenty pripevnené pomocou pájky. DPS obsahuje elektricky vodivé cesty ktoré navzájom prepájajú jednotlivé komponenty a taktiež privádzajú signál do a z pinov štítu cez ktoré je štít vsunutý do pinov Arduino dosky. Hlavné komponenty štítu sú potenciometer (slúžiaci na interakciu s prístrojom), ventilátor (ako zdroj prúdu vzduchu), korková loptička (levitujúci objekt), prehľadná plastová trubica (pre lepší vizuálny efekt) a laserový senzor vzdialenosti známy ako VL53L0X (na sledovanie polohy objektu).

Druhá kapitola sa zaoberá návrhom a tvorbou druhého a návrhom tretieho vydania zariadenia FloatShield. Jej prvá časť je zameraná na druhé vydanie zariadenia, pričom sú tu uvedené problémy ktoré boli zaznamenané pri práci s prvým vydaním zariadenia, a taktiež sú tu uvedené realizované riešenia týchto problémov v rámci tejto práce. Ide napríklad o využitie vložky so špeciálnym prierezom ktorá po nasadení do trubice a zároveň na ventilátor zariadenia FloatShield, vďaka svojmu tvaru pripomínajúcemu včelie plásty laminarizuje prúd vzduchu prúdiaci v trubici a tým znižuje tendenciu loptičky kmitať v dôsledku turbulentného prúdenia. Sú tu tiež uvedené špeciálne objekty ktoré boli vyrobené a otestované ako alternatíva loptičky so zámerom zvýšiť stabilitu levitujúceho objektu, ale aj potrebné zmeny ktoré bolo potrebné vykonať v elektrických cestách zariadenia aby sa dosiahla kompatibilita s 3.3 V doskami. Tieto zmeny sú najlepšie viditeľné na priložených nákresoch plošného spoja štítu a prislúchajúcej elektrickej

schémy. Nakoniec sú uvedené alternatívy upevnenia TOF senzora vzdialenosti, nakoľko pôvodné uchytenie sa ukázalo ako nespoľahlivé. Nové spôsoby zaručujú že sa senzor počas prebiehajúceho experimentu nepohne, aj napriek neustálym vibráciám ktoré vznikajú v dôsledku činnosti ventilátora.

V druhej časti kapitoly je navrhnuté tretie vydanie zariadenia FloatShield, nakoľko druhé vydanie sa ukázalo byť nepoužiteľné na získanie serióznych výsledkov z experimentov kvôli ventilátoru ktorý využíva. Konkrétny dôvod je vnútorné riadenie ktoré spomenutý ventilátor používa, vytvárajúce oneskorenie v odozve. To sa prejavuje približne sekundovou nečinnosťou ventilátora aj napriek zmene vstupného signálu, čo je v prípade nášho systému kde sa vzorkovanie pohybuje v desiatkach milisekúnd veľmi nežiaduce. Z tohto dôvodu je nevyhnutné aby tretie vydanie používalo iný ventilátor. Táto navrhovaná zmena je dôkladne zdokumentovaná slovne ako aj na priloženej elektrickej schéme a výkrese DPS.

Tretia kapitola je venovaná softvérovej podpore ktorá bola pre prístroj FloatShield vytvorená v rámci tejto práce. Táto podpora je realizovaná pomocou vytvorených funkcií pre všetky tri softvérové platformy ktoré zariadenie podporuje. Konkrétne ide o prostredia Arduino IDE, MATLAB a Simulink. V každom z nich sú definované knižnice, ktoré po zahrnutí do daného prostredia uľahčia prácu s prístrojom FloatShield. Jednou z ambícii projektu AutomationShield je dodržovať konvencie pomenúvania kľúčových premenných a funkcií naprieč všetkými prostrediami. Vo všetkých prostrediach potom možno nájsť funkcie s rovnakým názvom a rovnakým účelom. Napríklad ide o funkciu na inicializáciu zariadenia `begin()`, ktorá zabezpečí komunikáciu medzi počítačom a zariadením a taktiež medzi zariadením a senzorom vzdialenosti. Táto funkcia je nevyhnutná na operáciu zariadenia. Nasleduje kalibračná funkcia `calibrate()` ktorá získa medzné hodnoty merania zo senzora vzdialenosti, na základe ktorých potom ďalšie funkcie budú schopné prepočítavať svoj výstup. Potom ide napríklad o funkciu na získavanie meraní zo senzora `sensorRead()`, alebo na ovládanie výkonu ventilátora `actuatorWrite()` ktoré sú bežnou súčasťou mnohých experimentov. Mnoho špecializovaných funkcií bolo pre účely prístroja FloatShield vytvorených v prostredí MATLAB, nakoľko toto softvérové prostredie obsahuje množstvo užitočných nástrojov uľahčujúcich komplikované výpočty. Všetky vytvorené funkcie ako aj ich konkrétna implementácia formou C++ alebo MATLAB kódu sú uvedené v prílohách A a B.

Kapitola štyri sa zaoberá tvorbou matematického modelu a následnou identifikáciou parametrov modelu reprezentujúceho prístroj FloatShield. Na jeho základe je potom realizovaný proces odhadu pomocou algoritmu Kalmanovho filtra. Kapitola začína uvedením faktu, že prístroj FloatShield vykazuje silno nelineárne správanie. To možno pozorovať pri manuálnom ovládaní výkonu ventilátora pomocou potenciometra, a sledovaním ako bude korková loptička reagovať na tieto zmeny. Na základe tohto pokusu je možné si všimnúť, že zatiaľ čo ventilátor pri výkone (vyjadrenom v percentách) od 0% až do približne 35% nieje schopný zdvihnúť loptičku zo základu, pri výkone od 35% až do 100% loptička zaručene vzlietne až na samotný vrch trubice. Jediný rozdiel je v tomto prípade v rýchlosti stúpania – pri 35% je výstup postupný a pomalý zatiaľčo pri 100% je náhly. Podľa tohto pozorovania možno usúdiť, že ak je našim cieľom udržať loptičku v stabilnej pozícií, budeme na to pravdepodobne potrebovať výkon v okolí 40%. Spomenuté správanie je

priamym dôkazom nelinearity systému nakoľko nespĺňa princíp superpozície. Pri tvorbe modelu sa najskôr zvážilo, aké sily pôsobia na levitujúci objekt. Ide o silu gravitačnú a silu aerodynamického odporu ktorá vzniká v dôsledku stretu prúdu vzduchu a objektu. Na základe týchto dvoch síl, bola vytvorená pohybová rovnica prístroja FloatShield, vychádzajúca z druhého Newtonovho pohybového zákona. S pomocou takejto pohybovej rovnice boli následne postupne definované tri modely opisujúce systém FloatShield na základe vzťahu medzi jeho vstupmi a výstupmi, a to konkrétne vo forme linearizovanej prenosovej funkcie, linearizovaného stavového opisu a nelineárneho stavového opisu. Napriek tomu že takto bola získaná štruktúra modelu systému, jej jednotlivé parametre boli stále neznáme. Práve kvôli tomu ich bolo potrebné identifikovať na základe meraní. Na získanie vhodných dát bol použitý takzvaný PRBS signál, ktorý osciluje medzi dvoma krajnými hodnotami s pseudo-náhodne sa meniacou šírkou. Pomocou neho boli namerané dáta, na základe ktorých boli identifikované neznáme parametre v opisoch systému FloatShield, a tým boli získané tri kompletné varianty jeho modelu založené na druhom Newtonovom zákone. Pomocou linearizovaného stavového opisu bol vytvorený skript v MATLABe, v ktorom sa pomocou algoritmu Kalmanovho filtra odhadovali stavy systému FloatShield. Takýto odhad bol nevyhnutný, nakoľko z troch stavov ktorými je tento systém opísaný (výška loptičky, rýchlosť loptičky, rýchlosť prúdenia vzduchu v trubici), je známy iba prvý – získavaný pomocou senzora vzdialenosti. Schopnosť odhadovať aj ostatné stavy systému umožnila následné využitie pokročilých riadiacich algoritmov, konkrétne LQR a MPC, pričom obe závisia na stavovom opise systému.

Piata kapitola sa zaoberá príkladmi riadenia systému FloatShield ktoré boli v rámci tejto práce uskutočnené vo všetkých troch podporovaných softvérových prostrediach – Arduino IDE, MATLAB a Simulink. Kapitola začína zadefinovaním štruktúry týchto príkladov, aby sa čiastočne predišlo opakovaniu sa, nakoľko niektoré prvky všetkých príkladov sú spoločné, ako napríklad inicializácia zariadenia a následná kalibrácia senzora. V rámci tejto práce boli na prístroji FloatShield aplikované tri riadiace algoritmy – PID, LQR a MPC. Nasledujúce časti sa venujú uvedeniu čitateľa do základnej teórie za týmito algoritmami. Po zadefinovaní potrebnej teórie nasleduje opis jednotlivých príkladov vytvorených v jednotlivých softvérových prostrediach, spolu s uvedením a zhodnotením výsledkov riadenia pomocou daného algoritmu zobrazených na grafoch.

V druhej časti je opísaný PID regulátor ktorý počíta vstup do systému iba na základe chyby medzi meraním a referenčnou hodnotou. Tento algoritmus je pomerne jednoduchý avšak využíva zaujímavú myšlienku. Tým že pracuje s chybou v momentálnom kroku, s integrálom tejto chyby v čase a s deriváciou tejto chyby v čase, zjednodušene povedané je schopný počítať vstup do systému na základe minulosti, prítomnosti aj budúcnosti. Príspevky jednotlivých foriem tejto chyby do výsledného počítaného vstupu systému sú určované príslušnými konštantami regulátora. Nevýhodou PID je, že ním vypočítaný vstup do systému nerešpektuje procesné obmedzenia tohto vstupu. Napríklad, aj keď je systém schopný prijať vstup iba v percentuálnej forme v rozsahu 0%–100%, PID regulátor ľahko vypočíta hodnotu aj mimo tohto rozsahu. Toto je následne nutné vyriešiť v programe, kde je táto hodnota manuálne obmedzená aby bola použiteľná ako vstup do systému.

Tretia časť rozoberá lineárny kvadratický regulátor (LQR). Tento algoritmus je o niečo komplexnejší ako algoritmus PID regulátora, nakoľko LQR je závislý od stavového modelu

systému. Pri riadení využíva všetky stavy systému a teda je možné nastaviť referenčné hodnoty každému z nich, a tým riadiť systém ako celok (zatiaľ čo PID je primárne určený na riadenie iba výstupu systému). Na výpočet vstupu systému využíva rozdiel medzi referenčnými stavmi a súčasnými stavmi (chybu) a taktiež maticu zosilnenia `K`. Chyba je násobená maticou zosilnenia, a tak je vypočítaný vstup do systému. Táto matica sa získava pomocou matíc stavového opisu riešením takzvanej Ricattiho rovnice, na čo však je možné využiť funkciu MATLABu známu ako `dlqr()`. Nevýhodou tohto algoritmu je podobne ako pri PID, že neberie pri počítaní vstupu do systému ohľad na procesné obmedzenia, a teda vypočítaná hodnota musí byť manuálne obmedzená v programe. Navyše v prípade systému FloatShield nie sú merané a teda ani známe dva z troch stavov systému, kvôli čomu je nevyhnutný dodatočný algoritmus Kalmanovho filtra, pomocou ktorého tieto stavy budú odhadované.

Posledný spôsob riadenia je uvedený v štvrtej časti. Algoritmus prediktívneho riadenia (MPC) je najkomplexnejší z troch algoritmov implementovaných na zariadení FloatShield v rámci tejto práce. Podobne ako LQR závisí od stavového modelu systému a vyžaduje aby boli všetky jeho stavy známe, a teda aj v tomto prípade bolo potrebné využiť dodatočný algoritmus Kalmanovho filtra na odhad nemeraných stavov. Algoritmus MPC nezdieľa spoločnú nevýhodu PID a LQR, kde je potrebné počítaný vstup obmedzovať manuálne – v tomto prípade je možné zadefinovaním obmedzení pomocou špeciálnych matíc, predpísať regulátoru podmienky podľa ktorých bude riadiť systém. Konkrétne je možné obmedziť počítaný vstup do systému ale aj jednotlivé stavy systému, zadefinovaním ich horných a spodných hraníc. Regulátor potom bude počítať vstup tak, aby tieto podmienky dodržal. MPC je charakteristický tým, že na základe modelu predpovedá správanie systému a riadi ho tak, aby dosiahol kvalitné riadenie teraz, aj v budúcnosti. Keďže musí brať ohľad na budúcnosť aj na obmedzenia, je výpočet vstupu pomocou MPC algoritmu omnoho komplikovanejší ako len násobenie chyby špeciálnou maticou ako tomu bolo pri LQR alebo násobenie chyby konštantami ako tomu bolo pri PID. Výpočet vstupu je pri MPC založený na riešení optimalizačnej úlohy, kde sa hľadá minimum účelovej funkcie vzhľadom na premennú vstupu. Týmto spôsobom je získaný taký vstup do systému, ktorý zaručí minimálne "náklady" riadenia a tým optimálne riadenie z hľadiska využitia akčných vstupov. Riešenie optimalizačnej úlohy je pomerne náročné samo o sebe, no pri implementácií do vnorených systémov ako Arduino, sa často vyskytujú aj ďalšie komplikácie súvisiace s nedostatkom pamäte, nepostačujúcim výpočtovým výkonom alebo nedostupnosťou vhodného riešiča optimalizačnej úlohy.

V piatej časti je uvedený súhrn výsledkov z jednotlivých experimentov, a na ich základe sú tu jednotlivé algoritmy porovnané z hľadiska využitia vo vnorených systémoch. Na základe získaných výsledkov je možné vo všeobecnosti zjednodušene povedať, že na riadenie jednoduchších (ideálne lineárnych) systémov je najlepšia voľba PID, zatiaľ čo na riadenie komplexnejších systémov je lepšou možnosťou LQR a na riadenie veľmi zložitých systémov je dobré zvoliť MPC. V prípade LQR a MPC je však nevyhnutné poznať stavový opis riadeného systému. Navyše, najmä v prípade MPC, podľa požadovanej periódy vzorkovania kontrolovaného procesu môže byť limitujúcim faktorom jeho použitia výpočtový výkon vnoreného systému. Napriek tomu že podľa získaných výsledkov najlepšie riadenie v zmysle najrýchlejších a zároveň presných akčných zásahov dosiahol LQR, konkrétne výsledky vo veľkej miere závisia od naladenia daného algoritmu a teda nemožno robiť iba na ich základe závery. Riadenie systému FloatShield bolo vo všetkých prípadoch úspešné.

Appendix A

Arduino IDE Code

A.1 FloatShield.h

```
#ifndef FLOATSHIELD_H_
#define FLOATSHIELD_H_
#include "AutomationShield.h"
#include <Wire.h>
#include "lib/BasicLinearAlgebra/BasicLinearAlgebra.h"
#ifndef SHIELDRELEASE
  #define SHIELDRELEASE 1
#endif
#ifndef VL53L0X_h
  #include "lib/vl53l0x-arduino/VL53L0X.h"
#endif
#if SHIELDRELEASE == 1
  #define FLOAT_UPIN 3
  #define FLOAT_RPIN A0
#elif SHIELDRELEASE == 2
  #define FLOAT_UPIN 5
  #define FLOAT_RPIN A0
  #define FLOAT_YPIN 3
  #define FLOAT_RPM_CONST 68.1
#endif
#ifdef VL53L0X_h
class FloatClass {
  public:
    VL53L0X distanceSensor;
    void begin(void);
    void calibrate(void);
    void actuatorWrite(float);
    float referenceRead(void);
    float referenceReadAltitude(void);
    float sensorRead(void);
    float sensorReadAltitude(void);
    float sensorReadDistance(void);
    bool wasCalibrated(void);
    float returnMinDistance(void);
    float returnMaxDistance(void);
```

```
    float returnRange(void);
    #include "getGainLQ.inl"
    #include "getKalmanEstimate.inl"
#if SHIELDRELEASE == 2
    void actuatorWriteRPM(float);
    volatile unsigned int pulseCount;
    volatile bool pulseMeasured;
    void setSamplingPeriod(float);
    float sensorReadRPM(void);
#endif
  private:
    float _minDistance;
    float _maxDistance;
    float _range;
    bool _wasCalibrated;
    float _referenceValue;
    float _referencePercent;
    float _sensorValue;
    float _sensorPercent;
    float _ballAltitude;
#if SHIELDRELEASE == 2
    float _rpm;
    float _samplingPeriod = 25.0;
    float _nOfSamples = ceil(150.0 / _samplingPeriod);
    float _pulseCountToRPM = 15000.0 / (_samplingPeriod * _nOfSamples);
#endif
};
extern FloatClass FloatShield;
#endif
#endif
```

A.2 FloatShield.cpp

```
#include "FloatShield.h"
#ifdef VL53L0X_h
#if SHIELDRELEASE == 2
  void hallPeriodCounter(void) {
    FloatShield.pulseCount++;
    FloatShield.pulseMeasured = 1;
  }
#endif
void FloatClass::begin(void) {
  AutomationShield.serialPrint("FloatShield␣initialisation...");
#ifdef ARDUINO_ARCH_AVR
    Wire.begin();
  #if SHIELDRELEASE == 1
    analogReference(DEFAULT);
  #elif SHIELDRELEASE == 2
    analogReference(EXTERNAL);
  #endif
```

```
#elif ARDUINO_ARCH_SAM
  Wire1.begin();
#elif ARDUINO_ARCH_SAMD
  Wire.begin();
#endif
  distanceSensor.setTimeout(1000);
  if (!distanceSensor.init()) {
    AutomationShield.error("FloatShield␣failed␣to␣initialise!");
  }
  distanceSensor.setMeasurementTimingBudget(20000);
  distanceSensor.startContinuous();
  _minDistance = 17.0;
  _maxDistance = 341.0;
  _range = _maxDistance - _minDistance;
  _wasCalibrated = false;
#if SHIELDRELEASE == 2
  attachInterrupt(digitalPinToInterrupt(FLOAT_YPIN), hallPeriodCounter,
      CHANGE);
#endif
  AutomationShield.serialPrint("␣successful.\n");
}
void FloatClass::calibrate(void) {
  AutomationShield.serialPrint("Calibration...");
  float sum = 0.0;
  actuatorWrite(100.0);
  while (sensorReadDistance() > 100.0) {
    delay(100);
  }
  delay(1000);
  for (int i = 0; i < 100; i++) {
    sum += sensorReadDistance();
    delay(25);
  }
  _minDistance = sum / 100.0;
  sum = 0.0;
  actuatorWrite(0.0);
  while (sensorReadDistance() < 300.0) {
    delay(100);
  }
  delay(1000);
  for (int i = 0; i < 100; i++) {
    sum += sensorReadDistance();
    delay(25);
  }
  _maxDistance = sum / 100.0;
  _range = _maxDistance - _minDistance;
  _wasCalibrated = true;
  AutomationShield.serialPrint("␣sucessful.\n");
}
void FloatClass::actuatorWrite(float aPercent) {
  float mappedValue = AutomationShield.mapFloat(aPercent, 0.0, 100.0,
```

```
    0.0, 255.0);
  mappedValue = AutomationShield.constrainFloat(mappedValue, 0.0,
    255.0);
  analogWrite(FLOAT_UPIN, (int)mappedValue);
}
float FloatClass::referenceRead(void) {
  _referenceValue = (float)analogRead(FLOAT_RPIN);
  _referencePercent = AutomationShield.mapFloat(_referenceValue, 0.0,
    1023.0, 0.0, 100.0);
  return _referencePercent;
}
float FloatClass::referenceReadAltitude(void) {
  _referenceValue = (float)analogRead(FLOAT_RPIN);
  _referencePercent = AutomationShield.mapFloat(_referenceValue, 0.0,
    1023.0, 0.0, _range);
  return _referencePercent;
}
float FloatClass::sensorRead(void) {
  float readDistance = sensorReadDistance();
  _sensorPercent = AutomationShield.mapFloat(readDistance, _maxDistance
    , _minDistance, 0.0, 100.0);
  _sensorPercent = AutomationShield.constrainFloat(_sensorPercent, 0.0,
     100.0);
  return _sensorPercent;
}
float FloatClass::sensorReadAltitude(void) {
  _ballAltitude = sensorReadDistance();
  _ballAltitude = _maxDistance - _ballAltitude;
  _ballAltitude = constrain(_ballAltitude, 0, _maxDistance);
  return _ballAltitude;
}
float FloatClass::sensorReadDistance(void) {
  _sensorValue = (float)distanceSensor.readRangeContinuousMillimeters()
    ;
  return _sensorValue;
}
bool FloatClass::wasCalibrated(void) {
  return _wasCalibrated;
}
float FloatClass::returnMinDistance(void) {
  return _minDistance;
}
float FloatClass::returnMaxDistance(void) {
  return _maxDistance;
}
float FloatClass::returnRange(void) {
  return _range;
}
#if SHIELDRELEASE == 2
void FloatClass::actuatorWriteRPM(float rpm) {
  float calculatedPWM = rpm / FLOAT_RPM_CONST;
```

```
  calculatedPWM = AutomationShield.constrainFloat(calculatedPWM, 0.0,
     255.0);
  analogWrite(FLOAT_UPIN, (int)calculatedPWM);
}
void FloatClass::setSamplingPeriod(float sPeriod) {
  _samplingPeriod = sPeriod;
  _nOfSamples = ceil(150.0 / sPeriod);
  _pulseCountToRPM = 15000.0 / (sPeriod * _nOfSamples);
}
float FloatClass::sensorReadRPM(void) {
  static unsigned int sampleCounter = 0;
  if (pulseMeasured) {
    pulseMeasured = 0;
    sampleCounter++;
    if (sampleCounter == _nOfSamples) {
      _rpm = pulseCount * _pulseCountToRPM;
      pulseCount = 0;
      sampleCounter = 0;
    }
    return _rpm;
  } else
    pulseCount = 0;
    sampleCounter = 0;
  return 0;
}
#endif
FloatClass FloatShield;
#endif
```

A.3 getGainLQ.inl

```
template <int m, int n>
BLA::Matrix<m, n> getGainLQ(BLA::Matrix<n, n> &A, BLA::Matrix<n, m> &B,
    BLA::Matrix<n, n> &Q, BLA::Matrix<m, m> &R) {
  BLA::Matrix <m, n> K;
  K.Fill(0.0);
  BLA::Matrix <m, n> difference;
  K.Fill(-1.0);
  float differenceSum = -1.0;
  BLA::Matrix <n, n> P;
  for (int i = 0; i < n; i++) {
    for ( int j = 0; j < n; j++) {
      if (i == j) {
        P(i, j) = 1.0;
      } else {
        P(i, j) = 0.0;
      }
    }
  }
  BLA::Matrix <m, m> Inv ;
```

```
  Inv.Fill(0.0);
  while (differenceSum != 0.0) {
    differenceSum = 0.0;
    difference = K;
    P = (~(A - B * K) * P) * (A - B * K) + Q + ~K * R * K;
    Inv = (R + (~ B * P) * B);
    K = Invert (Inv) * ((~ B * P) * A);
    difference -= K;
    for (int i = 0; i < m; i++) {
      for ( int j = 0; j < n; j++) {
        differenceSum = differenceSum + abs(difference(i, j));
      }
    }
  }
  return K;
}
```

A.4 getKalmanEstimate.inl

```
template <int n>
void getKalmanEstimate(float* stateVector, float systemInput, float
   measuredOutput, BLA::Matrix<n, n> &A, BLA::Matrix<n, 1> &B, BLA::
   Matrix<1, n> &C, BLA::Matrix<n, n> &Q, BLA::Matrix<1, 1> &R) {
  static BLA::Matrix <n, 1> xEstimate;
  static BLA::Matrix <n, n> P;
  static BLA::Matrix <n, n> I;
  static bool wasInitialised = false;
  if (!wasInitialised) {
    for (int i = 0; i < n; i++) {
      if (C(i) == 1) {
        xEstimate(i) = measuredOutput;
      }
    }
    P.Fill(0);
    for (int i = 0; i < n; i++) {
      for ( int j = 0; j < n; j++) {
        if (i == j) {
          I(i, j) = 1.0;
        } else {
          I(i, j) = 0.0;
        }
      }
    }
    wasInitialised = true;
  }
  xEstimate = A * xEstimate + B * systemInput;
  P = A * P * ~A + Q;
  BLA::Matrix <n, 1> K = P * ~C / (C * P * ~C + R)(0);
  xEstimate = xEstimate + K * (measuredOutput - (C * xEstimate)(0));
  P = (I - K * C) * P;
```

```
  for (int i = 0; i < n; i++ ) {
    stateVector[i] = xEstimate(i);
  }
}
```

A.5 FloatShield_OpenLoop.ino

```
#include <FloatShield.h>
#include <Sampling.h>
unsigned long Ts = 25;
bool nextStep = false;
void setup() {
    Serial.begin(250000);
    FloatShield.begin();
    FloatShield.calibrate();
    Sampling.period(Ts*1000);
    Sampling.interrupt(stepEnable);
}
void loop() {
    if (nextStep) {
        step();
        nextStep = false;
    }
}
void stepEnable() {
    nextStep = true;
}
void step() {
    float potentiometerReference = FloatShield.referenceRead();
    float ballPositionInTube = FloatShield.sensorRead();
    FloatShield.actuatorWrite(potentiometerReference);
    Serial.print(potentiometerReference);
    Serial.print("␣");
    Serial.println(ballPositionInTube);
}
```

A.6 FloatShield_Identification.ino

```
#include <FloatShield.h>
#include <Sampling.h>
#define PRBS 0
#if PRBS
#include "prbsU.h"
int prbs;
#else
#include "aprbsU.h"
float aprbs;
#endif
```

```
unsigned long Ts = 25;
unsigned long k = 0;
bool nextStep = false;
bool realTimeViolation = false;
float y = 0.0;
float u = 0.0;
float stabilisedPower;
float stabilisationAltitude = 60;
float powerSpan = 1.5;
void setup() {
    Serial.begin(250000);
    FloatShield.begin();
    FloatShield.calibrate();
    Sampling.period(Ts*1000);
    PIDAbs.setKp(0.25);
    PIDAbs.setTi(5);
    PIDAbs.setTd(0.01);
    PIDAbs.setTs(Sampling.samplingPeriod);
    int stabilisationCounter=0;
    while(1) {
        y = FloatShield.sensorReadAltitude();
        u = PIDAbs.compute(stabilisationAltitude-y,30,100,30,100);
        FloatShield.actuatorWrite(u);
        if(y >= stabilisationAltitude-2.5 && y <= stabilisationAltitude
           +2.5) {
            stabilisationCounter++;
            if (stabilisationCounter==100) {
                stabilisedPower=u;
                break;
            }
        }
        delay(Ts);
    }
    Sampling.interrupt(stepEnable);
}
void loop() {
    if (nextStep) {
        step();
        nextStep = false;
    }
}
void stepEnable() {
    if(nextStep == true) {
        realTimeViolation = true;
        Serial.println("Real-time␣samples␣violated.");
        FloatShield.actuatorWrite(0.0);
        while(1);
    }
    nextStep = true;
}
void step() {
```

```
#if PRBS
    if(k>prbsU_length) {
        FloatShield.actuatorWrite(0.0);
        while(1);
    } else {
        prbs = pgm_read_word(&prbsU[k]);
        u = stabilisedPower+prbs*powerSpan;
    }
#else
    if(k>aprbsU_length) {
        FloatShield.actuatorWrite(0.0);
        while(1);
    } else {
        aprbs = pgm_read_float_near(&aprbsU[k]);
        u = stabilisedPower+aprbs*powerSpan;
    }
#endif
    y = FloatShield.sensorReadAltitude();
    FloatShield.actuatorWrite(u);
    Serial.print(y);
    Serial.print("␣");
    Serial.println(u);
    k++;
}
```

A.7 FloatShield_PID.ino

```
#include <FloatShield.h>
#include <Sampling.h>
#define MANUAL 0
unsigned long Ts = 25;
unsigned long k = 0;
bool nextStep = false;
bool realTimeViolation = false;
float r = 0.0;
float R[] = {65.0,50.0,35.0,45.0,60.0,75.0,55.0,40.0,20.0,30.0};
float y = 0.0;
float u = 0.0;
int T = 2400;
int i = 0;
#if SHIELDRELEASE == 1
  #define KP 0.25
  #define TI 5
  #define TD 0.01
#elif SHIELDRELEASE == 2
  #define KP 0.01
  #define TI 2
  #define TD 0.01
#endif
void setup() {
```

```
    Serial.begin(250000);
    FloatShield.begin();
    FloatShield.calibrate();
    Sampling.period(Ts*1000);
    PIDAbs.setKp(KP);
    PIDAbs.setTi(TI);
    PIDAbs.setTd(TD);
    PIDAbs.setTs(Sampling.samplingPeriod);
    while(1) {
#if MANUAL
        r = FloatShield.referenceRead();
#else
        r = R[0];
#endif
        y = FloatShield.sensorRead();
        #if SHIELDRELEASE == 1
          u = PIDAbs.compute(r-y,30,100,30,100);
        #elif SHIELDRELEASE == 2
          u = PIDAbs.compute(r-y,40,100,40,100);
        #endif
        FloatShield.actuatorWrite(u);
        if(y >= r*2/3) {
            break;
        }
        delay(Ts);
    }
    Sampling.interrupt(stepEnable);
}
void loop() {
    if (nextStep) {
        step();
        nextStep = false;
    }
}
void stepEnable() {
    if(nextStep == true) {
        realTimeViolation = true;
        Serial.println("Real-time␣samples␣violated.");
        FloatShield.actuatorWrite(0.0);
        while(1);
    }
    nextStep = true;
}
void step() {
#if MANUAL
    r = FloatShield.referenceRead();
#else
    if(i>(sizeof(R)/sizeof(R[0]))) {
        FloatShield.actuatorWrite(0.0);
        while(1);
    } else if (k % (T*i) == 0) {
```

```
        r = R[i];
        i++;
    }
#endif
    y = FloatShield.sensorRead();
    u = PIDAbs.compute(r-y,0,100,0,100);
    FloatShield.actuatorWrite(u);
    Serial.print(r);
    Serial.print("␣");
    Serial.print(y);
    Serial.print("␣");
    Serial.println(u);
    k++;
}
```

A.8 FloatShield_LQ.ino

```
#include <FloatShield.h>
#include <Sampling.h>
#define MANUAL 0
unsigned long Ts = 25;
unsigned long k = 0;
bool nextStep = false;
bool realTimeViolation = false;
float R[] = {210.0, 160.0, 110.0, 145.0, 195.0, 245.0, 180.0, 130.0,
   65.0,  95.0};
float y = 0.0;
float u = 0.0;
int T = 1200;
int i = 0;
BLA::Matrix<3, 3> A = {1, 0.02437, 0.00061, 0, 0.9502, 0.04721, 0, 0,
   0.89871};
BLA::Matrix<3, 1> B = {0.00011, 0.01321, 0.51677};
BLA::Matrix<1, 3> C = {1, 0, 0};
BLA::Matrix<3, 3> Q_Kalman = {5, 0, 0, 0, 1000, 0, 0, 0, 1000};
BLA::Matrix<1, 1> R_Kalman = {25};
BLA::Matrix<1, 4> K = {0.45815, 0.1997, 0.63981, -0.00252};
BLA::Matrix<4, 1> X = {0, 0, 0, 0};
BLA::Matrix<4, 1> Xr = {0, 0, 0, 0};
void setup() {
  Serial.begin(250000);
  FloatShield.begin();
  FloatShield.calibrate();
  Sampling.period(Ts * 1000);
  while (1) {
#if MANUAL
    Xr(0)  = FloatShield.referenceReadAltitude();
#else
    Xr(0)  = R[0];
#endif
```

```
    y = FloatShield.sensorReadAltitude();
    u = -(K * (X - Xr))(0);
    FloatShield.actuatorWrite(u);
    FloatShield.getKalmanEstimate(X, u, y, A, B, C, Q_Kalman,R_Kalman);
    X(3) = X(3) + (Xr(0) - X(0));
    if (y >= Xr(0) * 2 / 3) {
      break;
    }
    delay(Ts);
  }
  Sampling.interrupt(stepEnable);
}
void loop() {
  if (nextStep) {
    step();
    nextStep = false;
  }
}
void stepEnable() {
  if (nextStep == true) {
    realTimeViolation = true;
    Serial.println("Real-time␣samples␣violated.");
    FloatShield.actuatorWrite(0.0);
    while (1);
  }
  nextStep = true;
}
void step() {
#if MANUAL
  Xr(0)  = FloatShield.referenceReadAltitude();
#else
  if (i > (sizeof(R) / sizeof(R[0]))) {
    FloatShield.actuatorWrite(0.0);
    while (1);
  } else if (k % (T * i) == 0) {
    Xr(0)  = R[i];
    i++;
  }
#endif
  y = FloatShield.sensorReadAltitude();
  u = -(K * (X - Xr))(0);
  FloatShield.actuatorWrite(u);
  FloatShield.getKalmanEstimate(X, u, y, A, B, C, Q_Kalman,R_Kalman);
  X(3) = X(3) + (Xr(0) - X(0));
  Serial.print(Xr(0));
  Serial.print("␣");
  Serial.print(y);
  Serial.print("␣");
  Serial.println(u);
  k++;
}
```

A.9 FloatShield_MPC.ino

```
#include <FloatShield.h>
#include <Sampling.h>
#include "FloatShield_muAO-MPC/FloatShield_MPC.h"
#define MANUAL 0
unsigned long Ts = 25;
unsigned long k = 0;
bool nextStep = false;
bool realTimeViolation = false;
float Ref[] = {210.0, 160.0, 110.0, 145.0, 195.0, 245.0, 180.0, 130.0,
   65.0, 95.0};
float y = 0.0;
float u = 0.0;
int T = 2400;
int i = 0;
BLA::Matrix<3, 3> A = {1, 0.02437, 0.00061, 0, 0.9502, 0.04721, 0, 0,
   0.89871};
BLA::Matrix<3, 1> B = {0.00011, 0.01321, 0.51677};
BLA::Matrix<1, 3> C = {1, 0, 0};
BLA::Matrix<3, 3> Q_Kalman = {5, 0, 0, 0, 1000, 0, 0, 0, 1000};
BLA::Matrix<1, 1> R_Kalman = {25};
float X[4] = {0, 0, 0, 0};
float Xr[4] = {0, 0, 0, 0};
extern struct mpc_ctl ctl;
void setup() {
  Serial.begin(250000);
  FloatShield.begin();
  FloatShield.calibrate();
  Sampling.period(Ts * 1000);
  while (1) {
#if MANUAL
    Xr[0]  = FloatShield.referenceReadAltitude();
#else
    Xr[0]  = Ref[0];
#endif
    y = FloatShield.sensorReadAltitude();
    mpc_ctl_solve_problem(&ctl, X);
    u = ctl.u_opt[0];
    FloatShield.actuatorWrite(u);
    FloatShield.getKalmanEstimate(X, u, y, A, B, C, Q_Kalman,R_Kalman);
    X[3] = X[3] + (Xr[0] - X[0]);
    if (y >= Xr[0] * 2 / 3) {
      break;
    }
    delay(Ts);
  }
  Sampling.interrupt(stepEnable);
}
void loop() {
  if (nextStep) {
```

```
    step();
    nextStep = false;
  }
}
void stepEnable() {
  if (nextStep == true) {
    realTimeViolation = true;
    Serial.println("Real-time␣samples␣violated.");
    FloatShield.actuatorWrite(0.0);
    while (1);
  }
  nextStep = true;
}
void step() {
#if MANUAL
  Xr[0]  = FloatShield.referenceReadAltitude();
#else
  if (i > (sizeof(Ref) / sizeof(Ref[0]))) {
    FloatShield.actuatorWrite(0.0);
    while (1);
  } else if (k % (T * i) == 0) {
    Xr[0]  = Ref[i];
    i++;
  }
#endif
  y = FloatShield.sensorReadAltitude();
  mpc_ctl_solve_problem(&ctl, X);
  u = ctl.u_opt[0];
  FloatShield.actuatorWrite(u);
  FloatShield.getKalmanEstimate(X, u, y, A, B, C, Q_Kalman, R_Kalman);
  X[3] = X[3] + (Xr[0] - X[0]);
  Serial.print(Xr[0]);
  Serial.print("␣");
  Serial.print(y);
  Serial.print("␣");
  Serial.println(u);
  k++;
}
```

Appendix B

MATLAB Code

B.1 FloatShield.m

```
classdef FloatShield < handle
properties(Access = public)
    arduino;
    laserSensor;
end
properties(Access = private)
    FLOAT_UPIN = 'D3';
    FLOAT_RPIN = 'A0';
    minDistance = 17;
    maxDistance = 341;
end
methods(Access = public)
function begin(FloatShieldObject, aPort, aBoard)
    listOfLibraries = listArduinoLibraries();
    libraryIsPresent = sum(ismember(listOfLibraries,'Pololu/
       Pololu_VL53L0X'));
    if ~libraryIsPresent
        error('Pololu/Pololu_VL53L0X␣library␣was␣not␣detected␣by␣MATLAB
           !␣Use␣listArduinoLibraries()␣command␣to␣verify␣its␣presence.
           ')
    end
    FloatShieldObject.arduino = arduino(aPort, aBoard, 'Libraries', '
       Pololu/Pololu_VL53L0X','BaudRate',2000000)
    FloatShieldObject.laserSensor = addon(FloatShieldObject.arduino, '
       Pololu/Pololu_VL53L0X')
    beginSensor(FloatShieldObject.laserSensor)
    disp('FloatShield␣initialized.')
end
function calibrate(FloatShieldObject)
    actuatorWrite(FloatShieldObject, 100)
    while sensorReadDistance(FloatShieldObject) > 100
        pause(0.1)
    end
    pause(1)
    sum = 0;
```

```
    for i = 1:100
        sum = sum + sensorReadDistance(FloatShieldObject);
        pause(0.025)
    end
    FloatShieldObject.minDistance = sum / 100;
    actuatorWrite(FloatShieldObject, 0)
    while sensorReadDistance(FloatShieldObject) < 300
        pause(0.1)
    end
    pause(1)
    sum = 0;
    for i = 1:100
        sum = sum + sensorReadDistance(FloatShieldObject);
        pause(0.025)
    end
    FloatShieldObject.maxDistance = sum / 100;
    disp('FloatShield calibrated.')
end
function actuatorWrite(FloatShieldObject, percent)
    coercedInput = constrain(percent, 0, 100);
    writePWMDutyCycle(FloatShieldObject.arduino, FloatShieldObject.
       FLOAT_UPIN, (coercedInput / 100));
end
function reference = referenceRead(FloatShieldObject)
    reference = readVoltage(FloatShieldObject.arduino,
       FloatShieldObject.FLOAT_RPIN) * 100 / 5;
end
function minimalDistance = returnMinDistance(FloatShieldObject)
    minimalDistance = FloatShieldObject.minDistance;
end
function maximalDistance = returnMaxDistance(FloatShieldObject)
    maximalDistance = FloatShieldObject.maxDistance;
end
function ballPositionPercent = sensorRead(FloatShieldObject)
    distance = FloatShieldObject.sensorReadDistance();
    mapped = map(distance, FloatShieldObject.maxDistance,
       FloatShieldObject.minDistance, 0, 100);
    ballPositionPercent = constrain(mapped, 0, 100);
end
function ballAltitude = sensorReadAltitude(FloatShieldObject)
    reading = sensorReadDistance(FloatShieldObject);
    ballAltitude = FloatShieldObject.maxDistance - reading;
end
function rawSensorReading = sensorReadDistance(FloatShieldObject)
    rawSensorReading = readSensor(FloatShieldObject.laserSensor);
end
end
end
```

B.2 estimateKalmanState.m

```
function [stateEstimate, outputEstimate] = estimateKalmanState(
   systemInput, measuredOutput, A, B, C, Q, R, xEstimateIC)
if nargin < 7
    error('estimateKalmanState(u,y,A,B,C,Q,R) function expects at least
       seven input arguments!')
elseif nargin < 8
    xEstimateIC = zeros(length(A), 1);
end
persistent P xEstimate
if isempty(P)
    xEstimate = xEstimateIC;
    P = zeros(size(A));
end
xEstimate = A * xEstimate + B * systemInput;
P = A * P * A' + Q;
K = P * C' / (C * P * C' + R);
difference = measuredOutput - C * xEstimate;
xEstimate = xEstimate + K * difference;
P = (eye(size(K, 1)) - K * C) * P;
stateEstimate = xEstimate;
outputEstimate = C * stateEstimate;
end
```

B.3 plotResults.m

```
function plotResults(aFile)
if isa(aFile,'char')
data = load(aFile);
[fPath, fName, fExt] = fileparts(aFile);
switch lower(fExt)
    case '.mat'
        fileObject = matfile(aFile);
        fileDetails = whos(fileObject);
        r = data.(fileDetails.name)(:, 1);
        y = data.(fileDetails.name)(:, 2);
        u = data.(fileDetails.name)(:, 3);
    case '.txt'
        r = data(:, 1);
        y = data(:, 2);
        u = data(:, 3);
    otherwise
        error('Unexpected file extension: %s', fExt);
end
elseif isa(aFile,'double')
        data = aFile;
        r = data(:, 1);
        y = data(:, 2);
        u = data(:, 3);
```

```
end
k = 1:length(r);
maxValue = max([max(r), max(y), max(u)]);
minValue = min([min(r), min(y), min(u)]);
range = maxValue - minValue;
figure
plot(k, u, 'Color', [0.4660, 0.6740, 0.1880], 'LineWidth', 1)
hold on
plot(k, y, 'Color', [0.8500, 0.3250, 0.0980], 'LineWidth', 1)
stairs(k, r, 'Color', [0, 0.4470, 0.7410], 'LineWidth', 1)
hold off
xlim([k(1), k(end)])
ylim([minValue - range / 25, maxValue + range / 25])
title('Plot␣of␣experiment␣results')
legend('Input␣u(k)', 'Output␣y(k)', 'Reference␣r(k)', 'Location', '
   northwest')
xlabel('k')
ylabel('u(k)␣␣␣y(k)␣␣␣r(k)')
end
```

B.4 printBLAMatrix.m

```
function printBLAMatrix(varargin)
if nargin < 2
    error('Not␣enough␣input␣argumets␣provided.␣Function␣expects␣at␣
       least␣two␣arguments␣-␣printBLAMatrix(matrix,''matrixName'').')
end
if ~mod(nargin, 2)
    fileName = 'BLA_Matrices';
    nOfPairs = nargin / 2;
else
    fileName = varargin{nargin};
    nOfPairs = (nargin - 1) / 2;
end
fileHandle = fopen([fileName, '.h'], 'w');
fprintf(fileHandle, '//␣Header␣file␣storing␣BLA␣class␣matrices␣for␣
   Arduino␣IDE␣examples.\n');
fprintf(fileHandle, '//␣Automatically␣generated␣by␣the␣AutomationShield
   ␣library.\n');
fprintf(fileHandle, '//␣Visit␣www.automationshield.com␣for␣more␣
   information.\n');
fprintf(fileHandle, '//␣
   ============================================================\n');
fprintf(fileHandle, '\n');
for n = 1:nOfPairs
    matrix = varargin{2*n-1};
    matrixName = varargin{2*n};
    matrixSize = size(matrix);
    vector = matrix';
    vector = vector(:)';
```

```
    vector = round(vector, 5);
    fprintf(fileHandle, 'BLA::Matrix<%d, %d> %s%s', matrixSize(1),
       matrixSize(2), matrixName, ' = {');
    for i = 1:length(vector)
        fprintf(fileHandle, '%.5g', vector(i));
        if i < length(vector)
            fprintf(fileHandle, '%s', ', ');
        end
    end
    fprintf(fileHandle, '%s', '};');
    fprintf(fileHandle, '\n');
end
end
```

B.5 calculateCostFunctionMPC.m

```
function [H, G, F] = calculateCostFunctionMPC(A, B, np, Q, R, P)
[nx, nu] = size(B);
M = zeros(np*nx, nx);
for i = 1:np
    M(i*nx-nx+1:i*nx, :) = A^i;
end
N = zeros(nx*np, nu*np);
NN = zeros(nx*np, nu);
for i = 1:np
    NN(i*nx-nx+1:i*nx, :) = A^(i - 1) * B;
end
for i = 1:np
    N(i*nx-nx+1:end, i*nu-nu+1:i*nu) = NN(1:np*nx-(i - 1)*nx, :);
end
RR = kron(eye(np), R);
H = zeros(np*nu, np*nu);
G = zeros(np*nu, nx);
F = Q;
for i = 1:np - 1
    H = H + N((i - 1)*nx+1:i*nx, :)' * Q * N((i - 1)*nx+1:i*nx, :);
    G = G + N((i - 1)*nx+1:i*nx, :)' * Q * M((i - 1)*nx+1:i*nx, :);
    F = F + M((i - 1)*nx+1:i*nx, :)' * Q * M((i - 1)*nx+1:i*nx, :);
end
i = np;
H = H + N((i - 1)*nx+1:i*nx, :)' * P * N((i - 1)*nx+1:i*nx, :) + RR;
G = G + N((i - 1)*nx+1:i*nx, :)' * P * M((i - 1)*nx+1:i*nx, :);
F = F + M((i - 1)*nx+1:i*nx, :)' * P * M((i - 1)*nx+1:i*nx, :);
end
```

B.6 applyConstraintsMPC.m

```
function [Ac, b0, B0] = applyConstraintsMPC(p1, p2, p3, p4, p5, p6, p7)
switch nargin
    case 3
        nu = length(p2);
        Ac = [eye(p3*nu); -eye(p3*nu)];
        One = [];
        for i = 1:p3
            One = [One; eye(nu)];
        end
        b0 = [One * p2; -One * p1];
        B0 = [];
    case 5
        [nx, nu] = size(p5);
        M = zeros(p3*nx, nx);
        for i = 1:p3
            M(i*nx-nx+1:i*nx, :) = p4^i;
        end
        N = zeros(nx*p3, nu*p3);
        NN = zeros(nx*p3, nu);
        for i = 1:p3
            NN(i*nx-nx+1:i*nx, :) = p4^(i - 1) * p5;
        end
        for i = 1:p3
            N(i*nx-nx+1:end, i*nu-nu+1:i*nu) = NN(1:p3*nx-(i - 1)*nx,
               :);
        end
        Ac = [N; -N];
        B0 = [-M; M];
        One = [];
        for i = 1:p3
            One = [One; eye(nx)];
        end
        b0 = [One * p2; -One * p1];
    case 7
        [nx, nu] = size(p7);
        Acu = [eye(p5*nu); -eye(p5*nu)];
        One = [];
        for i = 1:p5
            One = [One; eye(nu)];
        end
        b0u = [One * p2; -One * p1];
        M = zeros(p5*nx, nx);
        for i = 1:p5
            M(i*nx-nx+1:i*nx, :) = p6^i;
        end
        N = zeros(nx*p5, nu*p5);
        NN = zeros(nx*p5, nu);
        for i = 1:p5
            NN(i*nx-nx+1:i*nx, :) = p6^(i - 1) * p7;
```

```
        end
        for i = 1:p5
            N(i*nx-nx+1:end, i*nu-nu+1:i*nu) = NN(1:p5*nx-(i - 1)*nx,
               :);
        end
        Acx = [N; -N];
        B0x = [-M; M];
        One = [];
        for i = 1:p5
            One = [One; eye(nx)];
        end
        b0x = [One * p4; -One * p3];
        Ac = [Acu; Acx];
        b0 = [b0u; b0x];
        B0 = [zeros(2*p5*nu, nx); B0x];
    otherwise
        disp('Invalid number of inputs!');
end
end
```

B.7 FloatShield_Ident_Greybox.m

```
clc; clear; close all;
load resultID.mat
Ts = 0.025;
data = iddata(y, u, Ts, 'Name', 'Experiment');
data = data(4530:5151);
data = detrend(data);
data.InputName = 'Fan Power';
data.InputUnit = '%';
data.OutputName = 'Ball Position';
data.OutputUnit = 'mm';
data.Tstart = 0;
data.TimeUnit = 's';
m = 0.00384;
r = 0.03;
cd = 0.74;
ro = 1.23;
A = 2 * pi * r^2;
g = 9.81;
k = 7.78;
tau = 0.14;
tau2 = 0.6;
h0 = data.y(1, 1);
dh0 = (data.y(2, 1) - data.y(1, 1)) / Ts;
v0 = 0;
model = 'nonlinear';
% model = 'linearSS';
% model = 'linearTF';
switch model
```

```
case 'nonlinear'
    FileName = 'FloatShield_ODE';
    Order = [1, 1, 3];
    Parameters = [m; cd; ro; A; g; k; tau];
    InitialStates = [h0; dh0; v0];
    Ts = 0;
    nlgr = idnlgrey(FileName, Order, Parameters, InitialStates, Ts,
        'Name', 'Nonlinear Grey-box Model');
    set(nlgr, 'InputName', 'Fan Power', 'InputUnit', '%', '
       OutputName', 'Ball Position', 'OutputUnit', 'mm', 'TimeUnit'
       , 's');
    nlgr = setpar(nlgr, 'Name', {'Ball Mass', 'Drag Coefficient', '
       Air Density', 'Exposed Area', ...
        'Gravitational Acceleration', 'Fan Gain', 'Fan Time
           Constant'});
    nlgr = setinit(nlgr, 'Name', {'Ball Position', 'Ball Speed', '
       Air Speed'});
    nlgr.Parameters(1).Fixed = true;
    nlgr.Parameters(2).Minimum = 0.3;
    nlgr.Parameters(2).Maximum = 1.0;
    nlgr.Parameters(3).Fixed = true;
    nlgr.Parameters(4).Fixed = true;
    nlgr.Parameters(5).Fixed = true;
    nlgr.Parameters(6).Minimum = 0;
    nlgr.Parameters(7).Minimum = 0;
    nlgr.InitialStates(3).Fixed = false;
    size(nlgr)
    opt = nlgreyestOptions('Display', 'on', 'EstCovar', true, '
       SearchMethod', 'Auto');
    opt.SearchOption.MaxIter = 50;
    model = nlgreyest(data, nlgr, opt);
case 'linearSS'
    r = 0.015;
    cd = 0.47;
    A = 2 * pi * r^2;
    vEq = sqrt(m*g/(0.5 * cd * ro * A));
    tau = vEq / (2 * g);
    alpha = 1 / tau;
    beta = 1 / tau2;
    gamma = k / tau;
    A = [0,      1,      0; ...
         0, -alpha,  alpha; ...
         0,      0,  -beta];
    B =     [0; ...
             0; ...
             gamma];
    C = [1, 0, 0];
    D = 0;
    K = zeros(3, 1);
    K(2) = 25;
    x0 = [h0; dh0; v0];
```

```
        disp('Initial guess:')
        sys = idss(A, B, C, D, K, x0, 0)
        sys.Structure.A.Free = [0, 0, 0; ...
                                0, 0, 0; ...
                                0, 0, 1];
        sys.Structure.B.Free = [0; ...
                                0; ...
                                1];
        sys.Structure.C.Free = false;
        sys.Structure.D.Free = false;
        sys.DisturbanceModel = 'estimate';
        sys.InitialState = 'estimate';
        Options = ssestOptions;
        Options.Display = 'on';
        Options.Focus = 'simulation';
        Options.EnforceStability = true;
        Options.InitialState = 'estimate';
        Options.SearchMethod = 'lsqnonlin';
        model = ssest(data, sys, Options);
    case 'linearTF'
        model = idproc('P2I');
        model.Structure.Kp.Value = k;
        model.Structure.Kp.Maximum = 10;
        model.Structure.Kp.Minimum = 5;
        model.Structure.Tp1.Value = tau;
        model.Structure.Tp1.Minimum = 0;
        model.Structure.Tp2.Value = tau2;
        model.Structure.Tp2.Minimum = 0;
        Opt = procestOptions;
        Opt.InitialCondition = 'estimate';
        Opt.DisturbanceModel = 'ARMA2';
        Opt.Focus = 'stability';
        Opt.Display = 'full';
        model = procest(data, model, Opt);
end
compare(data, model);
model
grid on
return
save FloatShield_GreyboxModel_Nonlinear.mat model
save FloatShield_GreyboxModel_LinearSS model
save FloatShield_GreyboxModel_LinearTF model
```

B.8 prbsGenerate.m

```
function prbsGenerate(varName,N,minu,maxu,B);
switch nargin
        case 2
            minu=-1;
            maxu=1;
            B=1;
        case 4
            B=1;
        case 5
        otherwise
            error('Check␣your␣arguments:␣variable␣name,␣length,␣upper␣
               limit,␣lower␣limit␣and␣unit␣band␣portion.')
end
band = [0, B];
levels = [minu maxu];
warning('off', 'Ident:dataprocess:idinput7')
prbs = idinput(N,'prbs',band,levels);
stairs(prbs)
axis([0,N,1.1*minu,1.1*maxu])
xlabel('Samples␣(-)')
ylabel('Amplitude␣(-)')
title('PRBS␣Signal')
vectorToC(prbs,varName,'int');
end
```

B.9 aprbsGenerate.m

```
function aprbsGenerate(varName, N, randomSeed, minu, maxu, B)
switch nargin
    case 2
        randomSeed = 1;
        minu = -1;
        maxu = 1;
        B = 1;
    case 3
        minu = -1;
        maxu = 1;
        B = 1;
    case 5
        B = 1;
    case 6
    otherwise
        error('Check␣your␣arguments:␣variable␣name,␣length,␣seed␣value,
           ␣upper␣limit,␣lower␣limit␣and␣unit␣band␣portion.')
end
band = [0, B];
levels = [minu, maxu];
warning('off', 'Ident:dataprocess:idinput7')
```

```
prbs = idinput(N, 'prbs', band, levels);
seed = randomSeed;
range = abs(maxu-minu);
for i = 1:length(prbs) - 1
    if prbs(i) == prbs(i+1)
        rng(seed);
        r = range * rand;
        if prbs(i) == maxu
            aprbs(i) = prbs(i) - r;
            aprbs(i+1) = prbs(i+1) - r;
        else
            aprbs(i) = prbs(i) + r;
            aprbs(i+1) = prbs(i+1) + r;
        end
    else
        if seed > 2^31
            seed = randomSeed;
        end
        seed = seed + 10;
        rng(seed);
        r = range * rand;
        if prbs(i+1) == maxu
            aprbs(i+1) = prbs(i+1) - r;
        else
            aprbs(i+1) = prbs(i+1) + r;
        end
    end
end
stairs(aprbs)
axis([0, N, minu - 0.1 * range, maxu + 0.1 * range])
xlabel('Samples␣(-)')
ylabel('Amplitude␣(-)')
title('APRBS␣Signal')
vectorToC(aprbs, varName, 'float');
end
```

B.10 FloatShield_KalmanFiltering.m

```
clc; clear; close all;
load resultID.mat
Ts = 0.025;
data = iddata(y, u, Ts, 'Name', 'Experiment');
data = data(4530:5151);
data = detrend(data);
load FloatShield_LinearSS_Discrete_Matrices_25ms.mat
Q_Kalman = diag([5, 1000, 1000]);
R_Kalman = 25;
clear estimateKalmanState
xEstimated = zeros(3, length(data.y));
yEstimated = zeros(1, length(data.y));
```

```
xIC = [data.y(1); (data.y(2) - data.y(1)) / Ts; 0];
for i = 1:length(data.y)
    [xEstimated(:, i), yEstimated(:, i)] = estimateKalmanState(data.u(i
       ), data.y(i), matA, matB, matC, Q_Kalman, R_Kalman, xIC);
end
velFromData = [diff(data.y)', 0] ./ Ts;
velFromEstimate = [diff(yEstimated), 0] ./ Ts;
t = (1:length(data.y)) * Ts;
figure
plot(t, data.y, 'y', 'LineWidth', 1.5)
hold on
plot(t, yEstimated, 'k', 'LineWidth', 1)
hold off
grid on
xlim([t(1), t(end)])
ylim([min(yEstimated) - 25, max(yEstimated) + 25])
legend('Original', 'Estimate')
xlabel('Time (s)'); ylabel('Ball Position (mm)')
title('Kalman filter test on identification data')
figure
plot(t,velFromData, 'b', 'LineWidth', 1.5)
hold on
plot(t,velFromEstimate, 'y', 'LineWidth', 1.5)
plot(t,xEstimated(2, :), 'k', 'LineWidth', 1.5)
xlim([t(1), t(end)])
ylim([min(velFromData) - 5, max(velFromData) + 5])
xlabel('Time (s)'); ylabel('Ball Velocity (mm/s)')
title('Ball velocity differentiated and estimated')
legend('Velocity differentiated from raw data', ...
    'Velocity differentiated from Kalman filtered data', ...
    'Velocity estimated by Kalman filter', 'Location', 'SouthEast')
grid on
figure
plot(t,xEstimated(2, :), 'b', 'LineWidth', 1.5)
xlabel('Time (s)'); ylabel('Ball Velocity (mm/s)')
xlim([t(1), t(end)])
ylim([min(xEstimated(2, :)) - 5, max(xEstimated(2, :)) + 5])
title('Ball velocity estimated by Kalman filter')
grid on
figure
plot(t,xEstimated(3, :), 'b', 'LineWidth', 1.5)
xlim([t(1), t(end)])
ylim([min(xEstimated(3, :)) - 5, max(xEstimated(3, :)) + 5])
xlabel('Time (s)'); ylabel('Air Velocity (mm/s)')
title('Air velocity estimated by Kalman filter')
grid on
return
printBLAMatrix(Q_Kalman, 'Q_Kalman', R_Kalman, 'R_Kalman', '
   BLA_Kalman_Matrices')
```

B.11 FloatShield_PID.m

```
clc; clear; close all;
FloatShield = FloatShield;
FloatShield.begin('COM4', 'UNO');
FloatShield.calibrate();
PID = PID;
Ts = 0.025;
k = 1;
nextStep = 0;
samplingViolation = 0;
R = [65, 50, 35, 45, 60, 75, 55, 40, 20, 30];
T = 2400;
i = 0;
response = zeros(length(R)*T, 3);
Kp = 0.25;
Ti = 5.0;
Td = 0.01;
PID.setParameters(Kp, Ti, Td, Ts);
while (1)
    r = R(1);
    y = FloatShield.sensorRead();
    u = PID.compute(r-y, 30, 100, 30, 100);
    FloatShield.actuatorWrite(u);
    if (y >= r * 2/3)
        break
    end
    pause(Ts)
end
tic
while (1)
    if (nextStep)
        if (mod(k, T*i) == 1)
            i = i + 1;
            if (i > length(R))
                FloatShield.actuatorWrite(0.0);
                break
            end
            r = R(i);
        end
        y = FloatShield.sensorRead();
        u = PID.compute(r-y, 0, 100, 0, 100);
        FloatShield.actuatorWrite(u);
        response(k, :) = [r, y, u];
        k = k + 1;
        nextStep = 0;
    end
    if (toc >= Ts * k)
        if (toc >= Ts * (k + 1))
            disp('Sampling␣violation␣has␣occured.')
            samplingViolation = 1
```

```
            FloatShield.actuatorWrite(0);
            break
        end
        nextStep = 1;
    end
end
response = response(1:k-1, :);
save responsePID response
disp('The␣example␣finished␣its␣trajectory.␣Results␣have␣been␣saved␣to␣"
   responsePID.mat"␣file.')
plotResults('responsePID.mat')
```

B.12 FloatShield_LQ.m

```
clc; clear; close all;
clear estimateKalmanState;
FloatShield = FloatShield;
FloatShield.begin('COM4', 'UNO');
FloatShield.calibrate();
Ts = 0.025;
k = 1;
nextStep = 0;
samplingViolation = 0;
Ref = [210,160,110,145,195,245,180,130,65,95];
T = 1200;
i = 0;
response = zeros(length(Ref)*T, 3);
X = [0; 0; 0; 0];
Xr = [0; 0; 0; 0];
load FloatShield_LinearSS_Discrete_Matrices_25ms
matAhat = [matA, zeros(3, 1); -matC, 1];
matBhat = [matB; 0];
Q = diag([1, 1, 1e7, 1e2]);
R = 1e7;
K = dlqr(matAhat, matBhat, Q, R);
while (1)
    Xr(1) = Ref(1);
    y = FloatShield.sensorReadAltitude();
    u = -K * (X - Xr);
    FloatShield.actuatorWrite(u);
    X(1:3) = estimateKalmanState(u, y, matA, matB, matC, Q_Kalman,
       R_Kalman);
    X(4) = X(4) + (Xr(1) - X(1));
    if (y >= Xr(1) * 2 / 3)
        break
    end
    pause(Ts)
end
tic
while (1)
```

```
    if (nextStep)
        if (mod(k, T*i) == 1)
            i = i + 1;
            if (i > length(Ref))
                FloatShield.actuatorWrite(0.0);
                break
            end
            Xr(1) = Ref(i);
        end
        y = FloatShield.sensorReadAltitude();
        u = -K * (X - Xr);
        FloatShield.actuatorWrite(u);
        X(1:3) = estimateKalmanState(u, y, matA, matB, matC, Q_Kalman,
           R_Kalman);
        X(4) = X(4) + (Xr(1) - X(1));
        response(k, :) = [Xr(1), y, u];
        k = k + 1;
        nextStep = 0;
    end
    if (toc >= Ts * k)
        if (toc >= Ts * (k + 1))
            disp('Sampling␣violation␣has␣occured.')
            samplingViolation = 1
            FloatShield.actuatorWrite(0);
            break
        end
        nextStep = 1;
    end
end
response = response(1:k-1, :);
save responseLQ response
disp('The␣example␣finished␣its␣trajectory.␣Results␣have␣been␣saved␣to␣"
   responseLQ.mat"␣file.')
plotResults('responseLQ.mat')
```

B.13 FloatShield_MPC.m

```
clc;clear;close all;
clear estimateKalmanState;
FloatShield = FloatShield;
FloatShield.begin('COM4', 'UNO');
FloatShield.calibrate();
Ts = 0.025;
k = 1;
nextStep = 0;
samplingViolation = 0;
Ref = [210,160,110,145,195,245,180,130,65,95];
T = 2400;
i = 0;
response = zeros(length(Ref)*T, 3);
```

```
X = [0; 0; 0; 0];
Xr = [0; 0; 0; 0];
load FloatShield_LinearSS_Discrete_Matrices_25ms
matAhat = [matA, zeros(3, 1); -matC, 1];
matBhat = [matB; 0];
uL = 0;
uU = 100;
Q = diag([1, 1, 1e7, 1e2]);
R = 1e7;
Np = 2;
[K, P] = dlqr(matAhat, matBhat, Q, R);
[H, G, F] = calculateCostFunctionMPC(matAhat, matBhat, Np, Q, R, P);
[Ac, b0] = applyConstraintsMPC(uL, uU, Np);
H = (H + H') / 2;
opt = optimoptions('quadprog', 'Display', 'none');
opt.SolverName='trust-region-reflective';
while (1)
    Xr(1) = Ref(1);
    y = FloatShield.sensorReadAltitude();
    u(:, 1) = quadprog(H, G*X(:, 1), Ac, b0, [], [], [], [], [], opt);
          FloatShield.actuatorWrite(u(1, 1));

    X(1:3) = estimateKalmanState(u(1, 1), y, matA, matB, matC, Q_Kalman
       , R_Kalman);
    X(4) = X(4) + (Xr(1) - X(1));
    if (y >= Xr(1) * 2 / 3)
        break
    end
    pause(Ts)
end
tic
while (1)
    if (nextStep)
        if (mod(k, T*i) == 1)
            i = i + 1;
            if (i > length(Ref))
                FloatShield.actuatorWrite(0.0);
                break
            end
            Xr(1) = Ref(i);
        end
        y = FloatShield.sensorReadAltitude();
        u(:, 1) = quadprog(H, G*X(:, 1), Ac, b0, [], [], [], [], [],
           opt);
        FloatShield.actuatorWrite(u(1, 1));
        X(1:3) = estimateKalmanState(u(1, 1), y, matA, matB, matC,
           Q_Kalman, R_Kalman);
        X(4) = X(4) + (Xr(1) - X(1));
        response(k, :) = [Xr(1), y, u(1, 1)];
        k = k + 1;
        nextStep = 0;
```

```
    end
    if (toc >= Ts * k)
        if (toc >= Ts * (k + 1))
            disp('Sampling violation has occured.')
            samplingViolation = 1
            FloatShield.actuatorWrite(0)
            break
        end
        nextStep = 1;
    end
end
response = response(1:k-1, :);
save responseMPC response
disp('The example finished its trajectory. Results have been saved to "
   responseMPC.mat" file.')
plotResults('responseMPC.mat')
```